

TURING 图灵程序设计丛书

松本行弘的程序世界

[日] 松本行弘 著
柳德燕 李黎明 译
夏倩 张文旭 译



人民邮电出版社
POSTS & TELECOM PRESS

版权信息

书名：松本行弘的程序世界

作者：松本行弘

译者：柳德燕，李黎明，夏倩，张文旭

ISBN：978-7-115-25507-5

本书由北京图灵文化发展有限公司发行数字版。版权所有，侵权必究。

您购买的图灵电子书仅供您个人使用，未经授权，不得以任何方式复制和传播本书内容。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

目录

[版权声明](#)

[推荐序](#)

[中文版序](#)

[前言](#)

[第 1 章 我为什么开发 Ruby](#)

[1.1 我为什么开发 Ruby](#)

- 第 2 章 面向对象
 - 2.1 编程和面向对象的关系
 - 2.2 数据抽象和继承
 - 2.3 多重继承的缺点
 - 2.4 两个误解
 - 2.5 Duck Typing 诞生之前
 - 2.6 元编程
- 第 3 章 程序块
 - 3.1 程序块的威力
 - 3.2 用块作循环
 - 3.3 精通集合的使用
- 第 4 章 设计模式
 - 4.1 设计模式 (1)
 - 4.2 设计模式 (2)
 - 4.3 设计模式 (3)
- 第 5 章 Ajax
 - 5.1 Ajax 和 JavaScript (前篇)
 - 5.2 Ajax 和 JavaScript (后篇)
- 第 6 章 Ruby on Rails
 - 6.1 MVC 和 Ruby on Rails
 - 6.2 开放类和猴子补丁
- 第 7 章 文字编码
 - 7.1 文字编码的种类
 - 7.2 程序中的文字处理
- 第 8 章 正则表达式
 - 8.1 正则表达式基础
 - 8.2 正则表达式的应用实例与“鬼车”
- 第 9 章 整数和浮点小数

- 9.1 深奥的整数世界
- 9.2 扑朔迷离的浮点小数世界
- 第 10 章 高速执行和并行处理
 - 10.1 让程序高速执行（前篇）
 - 10.2 让程序高速执行（后篇）
 - 10.3 并行编程
 - 10.4 前景可期的并行编程技术，Actor
- 第 11 章 程序安全性
 - 11.1 程序的漏洞与攻击方法
 - 11.2 用异常进行错误处理
- 第 12 章 关于时间的处理
 - 12.1 用程序处理时刻与时间
- 第 13 章 关于数据的持久化
 - 13.1 持久化数据的方法
 - 13.2 对象的保存
 - 13.3 关于 XML 的考察
- 第 14 章 函数式编程
 - 14.1 新范型——函数式编程
 - 14.2 自动生成代码
 - 14.3 内存管理与垃圾收集
 - 14.4 用 C 语言来扩展
 - 14.5 为什么要开源

版权声明

MATSUMOTO YUKIHIRO CODE NO SEKAI written by Yukihiro
Matsumoto Copyright © 2009 by Yukihiro Matsumoto. All rights reserved.
Originally published in Japan by Nikkei Business Publications, Inc.

本书中文简体字版由 Nikkei Business Publications 授权人民邮电出版社独家出版。未经出版者书面许可，不得以任何方式复制或抄袭本书内容。

版权所有，侵权必究。

推荐序

在流行的编程语言中，Ruby 比较另类，这是因为大多数编程语言的首要着眼点在于为解决特定的问题领域而设计语言，而 Ruby 的首要着眼点在于“人性化”，让程序员充分享受编程的乐趣。由于组织国内的 Ruby 会议，我曾经两次邀请松本行弘来中国。他是一位性格平和、对生活充满热爱的人，在演讲中也一再传递 `code for fun` 的宗旨，即编程语言不应该是冷冰冰地给机器阅读和执行的指令，而应该是让程序员编程的工作过程变成一种充满乐趣和享受的过程。而且，松本先生发明 Ruby 语言也是因为他对创造一种人性化的面向对象脚本语言的热爱。

程序员社区经常拿另外一个主流的面向对象脚本语言 Python 来和 Ruby 做对比。从全球范围来看，Python 的社区更大，应用更广泛，但 Ruby 的语法相对 Python 来说更强大和宽松，给程序员发挥的自由度更大，可以基于 Ruby 创建各个领域的 DSL，比方说 Ruby on Rails 就是一个基于 Ruby 的 Web 快速开发领域的 DSL。

总之，Ruby 语言的这种“人性化”以及给程序员很大编程自由度的气质奠定了整个 Ruby 社区的气质：热爱生活的程序员，追求编程的自由度，带点非主流的极客色彩。也正因为如此，Ruby 和基于 Ruby 的 Rails 得到了硅谷许许多多创业公司的青睐，有名者如 Twitter、Groupon、Hulu、github 等。而这种气质也很鲜明地体现在 Rails 框架的创建者 David Heinemeier Hansson 及其所在的 37signals 公司身上。

37signals 的 20 多位员工遍布全球，每周只上班四天，David Heinemeier Hansson 本人同时还是一位保时捷车队的职业赛车手。

当然，Ruby 并非只在非主流程程序员社区中流行，随着全球 IT 产业进入云计算时代，Ruby 也发挥着越来越大的作用。著名的 SAAS 厂商 salesforce 在 2010 年底以 2.1 亿美元收购了 PAAS 厂商 Heroku，并且在 2011 年 7 月聘请松本行弘担任 Heroku 首席架构师，开拓 Ruby 在云计算领域的应用。Heroku 本身就是一个完全采用 Ruby 架构的 PAAS 平台，同样支持 Ruby 的 PAAS 厂商还有 EngineYard、VMware 等。随着这些云计算厂商的努力，Ruby 必然在未来得到越来越广泛的应用。

我之前阅读了本书的部分章节，这本书实际上是松本行弘从一个编程语言设计者的角度去看待各种各样的流行编程语言，分析它们有哪些特点，以及 Ruby 编程语言是如何取舍的。Ruby 编程语言的设计本身大量地参考了一个更古老而著名的面向对象编程方法的开山之作 Smalltalk，而且从函数式编程语言鼻祖 LISP“偷师学艺”了不少好东西。程序员社区有个著名的说法：任何现代编程语言都脱胎于 Smalltalk 和 LISP，都与这两个编程语言有着似曾相识的特性，自 Smalltalk 和 LISP 诞生以来，编程语言领域可谓大势已定了。因此，集这两种编程语言诸多特点于一身的 Ruby 语言很值得编程爱好者去学习，而看看 Ruby 设计师是怎么设计 Ruby 语言的，则可以让你高屋建瓴地理解一些主流的编程语言。

范凯

ITeye 网站创始人，CSDN 产品总监

（图灵公司感谢李琳骁、常新居士等对本次重印勘误工作的贡献。）

中文版序

从年轻的时候开始，我就对编程语言有着极为浓厚的兴趣。比起“使用计算机干什么”这一问题，我总是一门心思想着“如何将自己的意图传达给计算机”。从这个意义上说，我认为自己是个“怪人”。但是，想选

择一个能让自己的工作变得轻松的编程语言，想编写一种让人用起来感到快乐的编程语言，一直是我梦寐以求的，这种迫切的心情恐怕不输于任何人。虽说是有点自卖自夸，但是 Ruby 确实给我带来了“快乐”，这一结果让我感到很满足。

让我感到惊奇的是，有很多人，包括那些没有我这么“怪”的人，都对这种快乐有着共鸣。Ruby 自 1995 年在互联网上公布以来，着实让世界各地的程序员都认识了它，共享着这种快乐，提高了软件开发的生产力。完全出乎我意料的是，世界各地的人，不管是东方还是西方，都极为欣赏 Ruby。在刚开始开发 Ruby 的时候，我想都没有想到过有这样的结果，程序员的感觉会超越人种、国籍、文化，有如此之多的共通之处。

现在，为世界各地的程序员所广泛接受的 Ruby，正带来一种新的文化。已经有越来越多的开发人员，在实践中果敢地施行着 Ruby 语言及其社区所追求的“对高生产力的追求”、“富有柔性的软件开发”、“对程序员人性的尊重”、“鼓起勇气挑战新技术”等原则。在 Ruby 以前，这些想法也都很好，却一直实践不起来。我相信，Ruby 的卓越之处，不仅在于语言能力，而且更重要的是引领了这种文化的践行。

本书在解说编程中的技术与原则时，不局限于表面现象，而是努力挖掘其历史根源，揭示其本质。虽然很多章节都以 Ruby 为题材，但这些原则对于 Ruby 以外的语言也行之有效。衷心希望大家能够实践本书中所讲述的各项原则，成为一个更好的开发人员。

松本行弘

2011 年 4 月 18 日

前言

本书的目的不是深入讲解哪种特定的技术，也没有全面讨论我所开发的编程语言 Ruby，而是从全局角度考察了与编程相关的各种技术。读者千万不要以为拿着这本书，就可以按图索骥地解决实际问题了。实

际上，最好把它看成是一幅粗略勾勒出了编程世界诸要素之间关系的“世界地图”。

每种技术、思想都有其特定的目的、渊源和发展进步的过程。本书试图换一个角度重新考察各种技术。如果你看过后能够感觉到“啊，原来是这样的呀！”或者“噢，原来这个技术的立足点在这里呀！”那么我就深感欣慰了。我的愿望就是这些知识能够激发读者学习新技术的求知欲。

本书的第 2 章到第 14 章，是在《日经 Linux》杂志于 2005 年 5 月到 2009 年 4 月连载的“松本编程模式讲坛”基础上编辑修改而成的。但实际上连载与最开始的设想并不一致，真正涉及“模式”的内容其实不多，倒是技术内幕、背景分析等内容占了主流。现在想来，大方向并没有错。

除了连载的内容之外，本书还记录了我对编程问题的新思考和新看法。特别是第 1 章“我为什么开发 Ruby”，针对“为什么是 Ruby”这一点，比其他杂志做了更加深入的解说。另外，在每章的末尾增加了一个小专栏。

对于连载的内容，因为要出成一本书，除修改了明显的错误和不合时代的部分内容之外，我力求每一章都独成一体、内容完整，同时也保留了连载时的风貌。通读全书，读者也许会感觉到有些话题或讲解是重复的，这一点敬请原谅。

我的本职工作是程序员，不能集中大段时间去写书，不过无论如何最后总算是赶出来了。非常感谢我的家人，她们在这么长时间里宽容着我这个情绪不稳的丈夫和父亲。

稿子写完了，书也出来了，想着总算告一段落了吧，而《日经 Linux》又要开始连载“松本行弘技术剖析”了，恐怕还要继续让家里人劳心。

松本行弘

2009 年 4 月于樱花季节过后的松江

第 1 章 我为什么开发 Ruby

1.1 我为什么开发 Ruby

Ruby 是起源于日本的编程语言。近年来，特别是因为其在 Web 开发方面的效率很高，Ruby 引起了全世界的关注，它的应用范围也扩展到了很多企业领域。

作为一门编程语言，Ruby 正在被越来越多的人所了解，而作为一介工程师的我，松本行弘，刚开始的时候并没有想过“让全世界的人都来用它”或者“这下子可以大赚一笔了”，一个仅仅是从兴趣开始的项目却在不知不觉中发展成了如今的样子。

当然了，那时开发 Ruby 并不是我的本职工作，纯属个人兴趣，我是把它作为一个自由软件来开发的。但是世事弄人，现在开发 Ruby 竟然变成我的本职工作了，想想也有些不可思议。

“你为什么开发 Ruby？”每当有人这样问我的时候，我认为最合适的回答应该就像 Linux 的开发者的 Linus Torvalds 对“为什么开发 Linux”的回答一样吧——

“因为它给我带来了快乐。”

当我还是一个高中生，刚刚开始学习编程的时候，不知何故，就对编程语言产生了兴趣。

周围很多喜欢计算机的人¹，有的是“想开发游戏”，有的是“想用它来做计算”，等等，都是“想用计算机来做些什么”。而我呢，则想弄明白“要用什么编程语言来开发”、“用什么语言开发更快乐”。

¹ 当时喜欢计算机的人当然还是少数。

高中的时候，我自己并不具备开发一种编程语言所必需的技术和知识，而且当时也没有计算机。但是，我看了很多编程语言类的书籍和杂志，知道了“还有像 Lisp 这样优秀的编程语言”、“Smalltalk 是做面向对象设计的”，等等，在这些方面我很着迷。上大学时就自然而然地

选修了计算机语言专业。10 年后，我通过开发 Ruby 实现了自己的梦想。

从 1993 年开始开发 Ruby 到现在已经过去 16 年了。在这么久的时间里，我从未因为设计 Ruby 而感到厌烦。开发编程语言真是一件非常有意思的事情。

1.1.1 编程语言的重要性

为什么会这么喜欢编程语言？我自己也说不清。至少，我知道编程语言是非常重要的。

最根本的理由是：语言体现了人类思考的本质。在地球上，没有任何超越人类智慧的生物，也只有人类能够使用语言。所以，正是因为语言，才造成了人类和别的生物的区别；正是因为语言，人和人之间才能传递知识和交流思想，才能做深入的思考。如果没有了语言，人类和别的动物也就不会有太大的区别了。

在语言学领域里，有一个 Sapir-Whorf 假说，认为语言可以影响说话者的思想。也就是说，语言的不同，造成了思想的不同。人类的自然语言是不是像这个假说一样，我不是很清楚²，但是我觉得计算机语言很符合这个假说。也就是说，程序员由于使用的编程语言不同，他的思考方法和编写出来的代码都会受到编程语言的很大影响。

² 在语言学中，Sapir-Whorf 假说好像是越来越站不住脚了。

也可以这么说，如果我们选择了好的编程语言，那么成为好程序员的可能性就会大很多。

20 年来一直被奉为名著的《人月神话》的作者 Frederick P. Brooks 说过：一个程序员，不管他使用什么编程语言，他在一定时间里编写的程序行数是一定的。如果真是这样，一个程序员一天可以写 500 行程序，那么不论他用汇编、C，还是 Ruby，他一天都应该可以写 500 行程序。

但是，汇编的 500 行程序和 Ruby 的 500 行程序所能做的事情是有天壤之别的。程序员根据所选择编程语言的不同，他的开发效率就会有十倍、百倍甚至上千倍的差别。

由于价格降低、性能提高，计算机已经很普及了。现在基本上各个领域都使用了计算机，但如果没有软件，那么计算机这个盒子恐怕一点用都没有了。而软件开发，就要求能够用更少的成本、更短的时间，开发出更多的软件。

需要开发的软件越来越多，开发成本却有限，所以对于开发效率的要求就很高。编程语言就成了解决这个矛盾的重要工具。

1.1.2 Ruby的原则

Ruby 本来是我因兴趣而开发的。因为对多种编程语言都很感兴趣，我广泛对比了各种编程语言，哪些特性好，哪些特性没什么用，等等，通过一一进行比较、选择，最终把一些好的特性吸纳进了 Ruby 编程语言之中。

如果什么特性都不假思索地吸纳，那么这种编程语言只会变成以往编程语言的翻版，从而失去了它作为一种新编程语言的存在价值。

编程语言的设计是很困难的，需要仔细斟酌。值得高兴的是，Ruby 的设计很成功，很多人都对 Ruby 给出了很好的评价。

那么，Ruby 编程语言的设计原则是什么呢？

Ruby 编程语言的设计目标是，让作为语言设计者的我能够轻松编程，进而提高开发效率。

根据这个目标，我制定了以下 3 个设计原则。

- 简洁性
- 扩展性
- 稳定性

关于这些原则，下面分别加以说明。

1.1.3 简洁性

以 Lisp 编程语言为基础而开发的商业软件 Viaweb 被 Yahoo 收购后，Viaweb 的作者 Paul Graham 也成了大富豪。最近他又成了知名的技术专栏作家，写了一篇文章就叫“简洁就是力量”³。

3 Paul Graham 目前是世界知名的天使投资人，其公司 Y Combinator 投资了很多极有前途的创业项目。Paul Graham 曾出版过两本 Lisp 专著，最新著作《黑客与画家》已经由人民邮电出版社出版。——编者注

他还撰写了很多倡导 Lisp 编程语言的文章。在这些文章中他提到，编程语言在这半个世纪以来是向着简洁化的方向发展的，从程序的简洁程度就可以看出一门编程语言本身的能力。上面提到的 Brooks 也持同样的观点。

随着编程语言的演进，程序员已经可以更简单、更抽象地编程了，这是很大的进步。另外随着计算机性能的提高，以前在编程语言里实现不了的功能，现在也可以做到了。

面向对象编程就是这样的例子。面向对象的思想只是把数据和方法看作一个整体，当作对象来处理，并没有解决以前解决不了的问题。

用面向对象记述的算法也一定可以用非面向对象的方法来实现。而且，面向对象的方法并没有实现任何新的东西，却要在运行时判定要调用的方法，倾向于增大程序的运行开销。即使是实现同样的算法，面向对象的程序往往更慢，过去计算机的执行速度不够快，很难允许像这样的“浪费”。

而现在，由于计算机性能大大提高，只要可以提高软件开发效率，浪费一些计算机资源也无所谓了。

再举一些例子。比如内存管理，不用的内存现在可用垃圾收集器自动释放，而不用程序员自己去释放了。变量和表达式的类型检查，在执行时已经可以自动检查，而不用在编译时检查了。

我们看一个关于斐波那契（Fibonacci）数的例子。图 1-1 所示为用 Java 程序来计算斐波那契数。算法有很多种，我们用最常用的递归算法来实现。

```
class Sample {
```

```

private static int fib (int n) {
    if (n<2) {
        return n;
    }
    else{
        return fib (n-2) +fib (n-1);
    }
}
public static void main (String[] argv) {
    System.out.println("fib(6)="+fib(6));
}
}

```

图 1-1 计算斐波那契数的 Java 程序

图 1-2 所示为完全一样的实现方法，它是用 **Ruby** 编程语言写的，算法完全一样。和 **Java** 程序相比，可以看到构造完全一样，但是程序更简洁。**Ruby** 不进行明确的数据类型定义，不必要的声明都可以省略。所以，程序就非常简洁了。

```

def fib (n)
  if n<2
    n
  else
    fib (n-2) +fib (n-1)
  end
end
print "fib (6) =", fib (6) , "\n"

```

图 1-2 计算斐波那契数的 Ruby 程序

算法的教科书总是用伪码来描述算法。如果像这样用实际的编程语言来描述算法，那么像类型定义这样的非实质代码就会占很多行，让人不能专心于算法。

如果可以把伪码中非实质的东西去掉，只保留描述算法的部分就直接运行，那么这种编程语言不就是最好的吗？**Ruby** 的目标就是成为开发效率高、“能直接运行的伪码式编程语言”。

1.1.4 扩展性

下一个设计原则是“扩展性”。编程语言作为软件开发工具，其最大的特征就是对要实现的功能事先没有限制。“如果想做就可以做到”，这听起来像小孩子说的话，但在编程语言的世界里，真的就是这么一回事。不管在什么领域，做什么处理，只要用一种编程语言编写出了程序，我们就可以说这种编程语言适用于这一领域。而且，涉及领域之广会远远超出我们当初的预想。

1999 年，关于 Ruby 的第一本书《面向对象脚本语言 Ruby》出版的时候，我在里面写道，“Ruby 不适合的领域”包括“以数值计算为主的程序”和“数万行的大型程序”。

但是几年后，规模达几万行、几十万行的 Ruby 程序被开发出来了。气象数据分析，乃至生物领域中也用到了 Ruby。现在，美国国家海洋和大气管理局（NOAA, National Oceanic and Atmospheric Administration）、美国国家航空和航天局（NASA, National Aeronautics and Space Administration）也在不同的系统中运用了 Ruby。

情况就是这样，编程语言开发者事先并不知道这种编程语言会用来开发什么，会在哪些领域中应用。所以，编程语言的扩展性非常重要。

实现扩展性的一个重要方法是抽象化。抽象化是指把数据和要做的处理都封装起来，就像一个黑盒子，我们不知道它的内部是怎么实现的，但是可以用它。

以前的编程语言在抽象化方面是很弱的，要做什么处理首先要了解很多编程语言的细节。而很多面向对象或者函数式的现代编程语言，都在抽象化方面做得很好。

Ruby 也不例外。Ruby 从刚开始设计时就用了面向对象的设计方法，数据和处理的抽象化提高了它的开发效率。我在 1993 年设计 Ruby 时，在脚本编程语言中采用面向对象思想的还很少，用类库方式来提供编程语言的就更少了。所以现在 Ruby 的成功，说明当时采用面向对象方法的判断是正确的。

Ruby 的扩展性不仅仅体现在这些方面。

比如 **Ruby** 以程序块这种明白易懂的形式给程序员提供了相当于 **Lisp** 高阶函数的特性，使“普通的程序员”也能够通过自定义来实现控制结构的高阶函数扩展。又比如已有类的扩展特性，虽然有一定的危险性，但是程序却可以非常灵活地扩展。关于这些面向对象、程序块、类扩展特性的内容，后面的章节还会详细介绍。

这些特性的共同点是，它们都表明了编程语言让程序员最大限度地获得了扩展能力。编程语言不是从安全性角度考虑以减少程序员犯错误，而是在程序员自己负责的前提下为他提供最大限度发挥能力的灵活性。我作为 **Ruby** 的设计者，也是 **Ruby** 的最初用户，从这种设计的结果可以看出，**Ruby** 看重的不是明哲保身，而是如何最大限度地发挥程序员自身的能力。

关于扩展性，有一点是不能忽视的，即“不要因为想当然而加入无谓的限制”。比如说，刚开始开发 **Unicode** 时，开发者想当然地认为 16 位（65 535 个字符）就足够容纳世界上所有的文字了；同样，**Y2K** 问题也是因为想当然地认为用 2 位数表示日期就够了才导致的。从某种角度说，编程的历史就是因为想当然而失败的历史。而 **Ruby** 对整数范围不做任何限定，尽最大努力排除“想当然”。

1.1.5 稳定性

虽然 **Ruby** 非常重视扩展性，但是有一个特性，尽管明知道它能带来巨大的扩展性，我却一直将其拒之门外。那就是宏，特别是 **Lisp** 风格的宏。

宏可以替换掉原有的程序，给原有的程序加入新的功能。如果有了宏，不管是控制结构，还是赋值，都可以随心所欲地进行扩展。事实上，**Lisp** 编程语言提供的控制结构很大一部分都是用宏来定义的。

所谓 **Lisp** 流，其语言核心部分仅提供极为有限的特性和构造，其余的控制结构都是在编译时通过用宏来组装其核心特性来实现的。这也就意味着，由于有了这种无与伦比的扩展性，只要掌握了 **Lisp** 基本语法 **S** 式（从本质上讲就是括号表达式），就可以开发出千奇百怪的语言。**Common Lisp** 的读取宏提供了在读取 **S** 式的同时进行语法变换的功能，这就在实际上摆脱了 **S** 式的束缚，任何语法的语言都可以用 **Lisp** 来实现。

那么，我为什么拒绝在 **Ruby** 中引入 **Lisp** 那样的宏呢？这是因为，如果在编程语言中引入宏的话，活用宏的程序就会像是用完全不同的专用编程语言写出来的一样。比如说 **Lisp** 就经常有这样的现象，活用宏编写的程序 **A** 和程序 **B**，只有很少一部分是共通的，从语法到词汇都各不相同，完全像是用不同的编程语言写的。

对程序员来说，程序的开发效率固然很重要，但是写出的程序是否具有高的可读性也非常重要。从整体来看，程序员读程序的时间可能比写程序的时间还长。读程序包括为理解程序的功能去读，或者是为维护程序去读，或者是为调试程序去读。

编程语言的语法是解读程序的路标。也就是说，我们可以不用追究程序或库提供的类和方法的详细功能，但是，“这里调用了函数”、“这里有判断分支”等基本的“常识”在我们读程序时很重要。

可是一旦引入了宏定义，这一常识就不再适用了。看起来像是方法调用，而实际上可能是控制结构，也可能是赋值，也可能有非常严重的副作用，这就需要我们去查阅每个函数和方法的文档，解读程序就会变得相当困难。

当然了，我知道世界上有很多 **Lisp** 程序员并不受此之累，他们正是通过面向特定程序定制语言而最大限度地提高了开发效率。不过在我个人看来，他们只是极少数的一部分程序员。

我相信，作为在世界上广泛使用的编程语言，应该有稳定的语法，不能像随风飘荡的灯芯那样闪烁不定。

1.1.6 一切皆因兴趣

当然，**Ruby** 不是世界上唯一的编程语言，也不能说它是最好的编程语言。各种各样的编程语言可以在不同的领域中应用，各有所长。我自己以及其他 **Ruby** 程序员，用 **Ruby** 开发效率很高，所以觉得 **Ruby**“最为得心应手”。当然，用惯了 **Python** 或者 **Lisp** 的程序员，也会觉得那些编程语言是最好的。

不管怎么说，编程语言存在的目的是让人用它来开发程序，并且尽量能提高开发效率。这样的话，才能让人在开发中体会到编程的乐趣。

我在海外讲演的时候，和很多人交流过使用 Ruby 的感想，比较有代表性的是：“用 Ruby 开发很快乐，谢谢！”

是啊，程序开发本来就是一件很快乐、很刺激和很有创造性的事情。想起中学的时候，用功能不强的 BASIC 编程语言开发，当时也是很快乐的。当然，工作中会有很多的限制和困难，编程也并不都是一直快乐的，这也是世之常情。

Ruby 能够提供很高的开发效率，让我们在工作中摆脱很多困难和烦恼，这也是我开发 Ruby 的目的之一吧。

第 2 章 面向对象

2.1 编程和面向对象的关系

所谓编程，就是把工作的方法告诉计算机。但是，计算机是没有思想的，它只会简单地按照我们说的去做。计算机看起来功能很强大，其实它也仅仅只会做高速计算而已。如果告诉它效率很低的方法，它也只是简单机械地去执行。所以，到底是最大程度地发挥计算机的能力，还是扼杀它的能力，都取决于我们编写的程序了。

程序员让计算机完全按照自己的意志行事，可以说是计算机的“主宰”。话虽如此，但世人多认为程序员是在为计算机工作。

不，不只是一般人，很多计算机业内人士也是这样认为的，甚至比例更高。难道因为是工作，所以就无可奈何了吗？

2.1.1 颠倒的构造

如果仔细想想，就会感到很不可思议。为什么程序员非要像计算机的奴隶一样工作呢？我们到底是从什么时候开始放弃了主宰计算机这个念头的呢？

我想，其中的一个原因是“阿尔法综合征”。阿尔法综合征是指在饲养宠物狗的时候，宠物狗误解了一直细心照顾它的主人的地位，反而感

觉到它自己是主人，比主人更了不起。

计算机也不是好伺候的。系统设计困难重重，程序有时也会有错误。一旦有规格变更，程序员就要动手改程序，程序有了错误，也需要一个个纠正过来。

所以在诸如此类烦琐的工作中，就会发生所谓的“逆阿尔法综合征”现象，主从关系颠倒，程序员沦为“计算机的奴隶”，说得客气一些，也顶多能算是“计算机的看门狗”。难道这是人性使然？

不，不要轻易放弃。人是万物之灵，比计算机那玩意儿要聪明百倍，当然应该摆脱计算机奴隶的地位，把工作都推给机器来干，自己尽情享受轻松自在。因此，我们的目标就是让程序员夺回主动权！

程序员如果能够充分利用好计算机所具有的高速计算能力和信息处理能力，有可能会从奴隶摇身一变，“像变戏法一样”完成工作，实现翻天覆地的大逆转。

但是，要想赢得这场夺回主动权的战争，“武器”是必需的。那就是本书要讲解的“语言”和“技术”。

Ruby 的安装

读者中恐怕有不少人是初次安装 Ruby，所以这里再介绍一下 Ruby 的安装方法。在我写这本书的时候，Ruby 的版本是 1.9.1。在我平时使用的 Debian GNU/Linux 操作系统中，用下面的方法来安装 Ruby。

```
$ apt-get install ruby
```

其他的 Linux 操作系统大多也提供了 Ruby 的开发包。

在 Windows 操作系统中安装 Ruby 时，直接点击安装文件就可以了。从下面的网站可以下载安装程序：

<http://rubyinstaller.rubyforge.org>。

如果从 Ruby 源程序来编译安装的话，可以从下面的网站来下载 Ruby 源程序包（tarball）：<http://ruby-lang.org>。

编译和安装的方法如下。

```
$ tar zxvf ruby-1.9.1-p0.tar.gz
$ cd ruby-1.9.1-p0
$ ./configure
$ make
$ su
# make install
```

2.1.2 主宰计算机的武器

程序员或者将要成为程序员的人，如果成了计算机的奴隶，那是十分不幸的。为了能够主宰计算机，必须以计算机的特性和编程语言作为武器。

编程语言是描述程序的方法。目前有很多种编程语言，有名的有 BASIC、FORTRAN、C、C++、Java、Perl、PHP、Python、Ruby 等。

从数学的角度来看，几乎所有的编程语言都具备“图灵完备”¹ 的属性，无论何种编程语言都可以记述等价的程序，但这并不是说选择什么样的编程语言都一样。每种编程语言都有自己的特征、属性，都各有长处和短处、适合的领域和不适合的领域。写程序的难易程度（生产力）也有很大的不同。

¹ 图灵完备指在可计算性理论中，编程语言或任意其他的逻辑系统具有等同于通用图灵机的计算能力。换言之，此系统可与通用图灵机互相模拟。这个词源于引入图灵机概念的数学家阿兰·图灵（Alan Turing）。

有研究表明，开发程序时用的编程语言和生产力并没有关系，不论用什么编程语言，一定时间内程序的开发规模（在一定程度上）是相当的。

还有一些研究表明，对于同样的任务，程序规模会因为开发时选取的编程语言和库而相差数百倍，甚至数千倍。所以如果选用了合适的编程语言，那么你的能力就可能增长数千倍。

但是不论什么都是有代价的。比如效率高的开发环境，在执行时效率往往会很低。还有很多领域需要人们想尽办法去提高速度。在这里，因为我们在讨论如何主宰计算机，所以尽可能地选择让人轻松的编程

语言。基于这个观点，本书用 Ruby 编程语言来讲解。当然，Ruby 是我设计的，讲解起来相对也就容易点。

Ruby 是面向对象的编程语言，具有简洁和一致性。开发 Ruby 的宗旨是用它可以轻松编程。

Ruby 的运行环境多种多样，包括 Linux 及 UNIX 系列操作系统、Windows、MacOS X 等各种平台，很多系统上都有 Ruby 的软件包²。当然，如果有 C 编译器，也可以从源代码来安装 Ruby。

² 比如 MacOS X 10.5 中标准搭载了 Ruby 1.8.6。

2.1.3 怎样写程序

使用编程语言写好程序是有技巧的。在本书中，将会介绍表 2-1 中列出的编程技巧。

表2-1 本书讲解的主要编程技术

编程风格
算法
数据结构
设计模式
开发方法

表中的编程风格指的是编程的细节，比如变量名的选择方法、函数的写法等。

算法是解决问题的方法。现实中各种算法都已经广为人知了，所以编程时的算法也就是对这些技巧的具体应用。

有很多算法，如果单自己去想是很难想出来的。比方说数组的排序就有很多的算法，如果我们对这些算法根本就不了解，那么要想做出高速排序程序会很困难。算法和特定的数据结构关系很大。所以有一位计算机先驱曾经说过：“程序就是算法加数据结构。”³

3 *Algorithms + Data Structures = Programs* , Niklaus Wirth 著。Wirth 是在 1971 年开发了 Pascal 编程语言的计算机学者。

设计模式是指设计软件时，根据以前的设计经验对设计方法进行分类。算法和数据结构从广义上来说也是设计模式的一种分类。有名的分类（设计模式）有 23 种⁴。

4 《设计模式：可复用面向对象软件的基础》，Erich Gamma 等著，机械工业出版社出版。

开发方法是指开发程序时的设计方法，指包括项目管理在内的整个程序开发工程。小的软件项目可能不是很明显，在大的软件项目中，随着开发人员的增加，整个软件工程的开发方法就很重要。

2.1.4 面向对象的编程方法

下面，我们来看看 Ruby 的基本原理——面向对象的设计方法。面向对象的设计方法是 20 世纪 60 年代后期，在诞生于瑞典的 Simula 编程语言中最早开始使用的。Simula 作为一种模拟语言，对于模拟的物体，引入了对象这种概念。比如说对于交通系统的模拟，车和信号就变成了对象。一辆辆车和一个个信号就是一个个对象，而用来定义这些车和信号的，就是类。

此后，从 20 世纪 70 年代到 80 年代前期，美国施乐公司的帕洛阿尔托研究中心（PARC）开发了 Smalltalk 编程语言。从 Smalltalk-72、Smalltalk-78 到 Smalltalk-80，他们开发完成了整个 Smalltalk 系列。Smalltalk 编程语言对近代面向对象编程语言影响很大，所以把它称为面向对象编程语言之母也不为过。

在这之后，受 Simula 影响比较大的有 C++ 编程语言，再以后还有 Java 编程语言，而现在大多数编程语言使用的都是面向对象的设计方法。

如今，面向对象的设计思想已经相当重要且深入人心了，以后它的地位和重要性也应该不会降低。所以在学习编程语言时，对面向对象设计思想的理解就非常重要。

但是，很多程序员觉得面向对象的设计思想很难，不容易理解，所以在本章中，我们将详细介绍一下面向对象的设计方法。

2.1.5 面向对象的难点

面向对象的难点在于，虽然有关于面向对象的说明和例子，但是面向对象具体的实现方法却不是很明确。

面向对象这个词本身是很抽象的，越抽象的东西，人们就越难理解。并且对于面向对象这个概念，如果没有严密的定义，不同的人就会有不同的理解。

这里，我们暂时回避一下“面向对象”的整体概念这一问题，首先集中说明“面向对象编程”。

至于“好像是听明白了，还是不会使”这一点，原因可能在于平易的比喻和实际编程之间差距太大。这里，我们选择 **Ruby** 这种简单易用的面向对象编程语言，希望能够拉近比喻和实例之间的距离。

另外很重要的一点，面向对象编程语言有很多种类，也有很多技巧。一下子全部理解是很困难的，我们分别加以说明。

我认为面向对象编程语言中最重要的技术是“多态性”。我们就先从多态性说起吧。

2.1.6 多态性

多态性，英文是 **polymorphism**，其中词头 **poly-**表示复数，**morph** 表示形态，加上词尾 **-ism**，就是复数形态的意思，我们称它为多态性。

换个说法，多态就是可以把不同种类的东西当做相同的东西来处理。

只从字面上分析不容易理解，举例说明一下。

看看图 2-1 所示的 3 个箱子。每个箱子都有不同的盖子。一个是一般的盖子，一个是带锁的盖子，一个是带有彩带的盖子。因为箱子本身非常昂贵，所以每个箱子都有专人管理，如果要从箱子里取东西，要由管理人员去做。

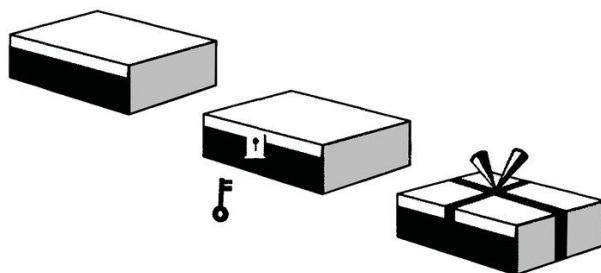


图 2-1 操作对象是 3 个箱子，分别是盖着盖子的箱子、加了锁的箱子、系着彩带的箱子

打开 3 个箱子的方法都不同，但如果发出同样的打开箱子的命令，3 个人会用自己的方法来打开自己的箱子。因此，3 个箱子虽然各有不同，但它们同样“都是箱子，可以打开盖子”。这就是多态性的本质。

在编程中，“打开箱子”的命令，我们称之为消息；而打开不同箱子的具体操作，我们称之为方法。

2.1.7 具体的程序

上面例子的程序如图 2-2 所示。

```
# 用变量 box1、box2、box3 代表 3 个箱子

box_open(box1) # 表示打开箱子
box_open(box2) # 表示开锁，打开箱子
box_open(box3) # 表示解开彩带，打开箱子
```

图 2-2 例子的程序

`box_open` 是打开箱子的方法，相当于前面所说的“管理员”。调用 `box_open` 这个方法时，方法会根据参数（箱子的种类）的不同做相应的处理。你只要说“打开箱子”，箱子就真的被打开了。这种“根据对象不同类型而进行适当地处理”就是多态性的基本内容。

但只有图 2-2 还不够。我们来考虑一下如何定义 `box_open` 这个方法吧。如果只是单纯地实现这个方法，也许就会写成图 2-3 的样子。

```
def box_open(box)

    # 判断 box 类型的方法
```

```
    if box_type(box)=="plain"
        puts("打开箱子")
    elsif box_type(box)=="lock"
        puts("开锁，打开箱子")
    elsif box_type(box)=="ribbon"
        puts("解开彩带，打开箱子")
    else
        puts("不知道打开箱子的方法")
    end
end
```

图 2-3 图 2-2 例子的 `box_open` 方法的内容

但是，图 2-3 所示的处理并不能令人满意。如果要增加箱子种类，这个方法中的代码就要重写，而且如果还有其他类似于 `box_open`、需要根据箱子类型来做不同处理的方法，那么需要修改的地方就越来越多，追加箱子种类就会变得非常困难。

程序修改得越多，出错的可能性也就越大，结果可能是程序本身根本就动不起来了。

像这样的修改本来就不该直接由人来做。根据数据类型来进行合适的处理（调用合适的方法），本来就应该是编程语言这种工具应该完成的事。只有实现了这一点，才能称为真正的多态。

为此，我们修改一下图 2-2 的程序，来看看真正的多态是如何工作的。

图 2-4 的程序把参数移到了前头，并增加了一个“.”。这行代码可以理解为“给前面式子的值发送 `open` 消息”。也就是说，它会“根据前面式子的值，调用合适的 `open` 方法”。这就是利用了多态性的调用方法。

```
#用变量 `box1`、`box2`、`box3` 代表 3 种箱子

box1.open() # 表示打开箱子
box2.open() # 表示开锁，打开箱子
box3.open() # 表示解开彩带，打开箱子
```

图 2-4 图 2-2 例子程序的改良版，改变了参数调用的方法

图 2-4 程序中的各种处理方法的定义如图 2-5 所示。

```
#用变量box1、box2、box3 代表3 种箱子

def box1.open()
  puts("打开箱子")
end

def box2.open()
  puts("开锁，打开箱子")
end

def box3.open()
  puts("解开彩带，打开箱子")
end
```

图 2-5 方法的定义

图 2-5 的程序定义了 3 种箱子：**box1**、**box2**、**box3**，表示“打开箱子”的不同方法。

比较图 2-5 和图 2-3 的程序可以看到，程序中不再有直白的条件判断，非常简单明了。即使在图 2-5 中程序增加一种新的箱子，比如“横向滑动之后打开箱子”，也不需要原来的程序做任何修改。不需要修改，当然也就没有因修改而出错的危险。

2.1.8 多态性的优点

前面说明了多态性，那么它到底有什么好处呢？

首先，各种数据可以统一地处理。多态性可以让程序只关注要处理什么（**What**），而不是怎么去处理（**How**）。

其次，是根据对象的不同自动选择最合适的方法，而程序内部则不发生冲突。不管调用有锁的箱子，还是系着彩带的箱子，它们都能自动处理，不用担心调用中会发生错误，这样就会减轻程序员的负担。

再次，如果有新数据需要对应处理的话，通过简单的追加就可以实现了，而不需要改动以前的程序，这就让程序具备了扩展性。

综上所述，多态性提高了开发效率，所以说，面向对象技术最重要的一个概念应该是多态性。

相关的 Ruby 语法

为了让读者能理解本书中的程序例子，这里简单说明一下 Ruby 语法。

首先，以“#”开始的行是注释行，注释的内容随便是什么都可以。

```
# 这一行是注释行
```

条件判断用 **if** 语句。

```
if 条件
  处理代码
elsif 条件
  处理代码
else
  处理代码
end
```

具体的程序如图 2-6 所示。

```
if box_type(box) == "plain"
  puts("打开箱子")
elsif box_type(box) == "lock"
  puts("用钥匙打开箱子")
elsif box_type(box) == "ribbon"
  puts("解开彩带，打开箱子")
else
  puts("不知道打开箱子的方法")
end
```

图 2-6 条件判断程序

当第一个条件成立的时候，就执行第一段处理代码；当第二个条件成立的时候，就执行第二段处理代码；而当所有条件都不成立的时候，就执行 **else** 下面的处理代码。如果处理代码由多条语句并列构成，不需要用“{ }”括起来，而是用 **elsif** 或者 **end** 等保留词来分隔，这一点也许会让你觉得耳目一新。

在 Ruby 的 **if** 语句中，**elsif** 部分可以重复出现任意次。当然也可以是 0 次，这时候 **elsif** 是可以省略的。**else** 同样也是可以省略的。

对于"plain"来说，"" 中的是字符串。与数值一样，字符串也是能直接写在程序里的数据。在 Ruby 中，这些数据都是对象，我们将在以后的章节中详细说明。

像 **box** 这样，以小写英文字母开头的是变量。这个例子中已事先设置好了变量的值。像其他的编程语言一样，变量的赋值语句是

```
变量 = 值
```

用来初始化变量。

要判断两个表达式的值是否一样，可以使用“==”运算符。

```
表达式 == 表达式
```

请注意，在赋值语句中是用一个等号，而判断两个表达式是否相等则是用两个等号。这跟 Java 或 C 等许多语言中的用法也都是是一样的。

后面有小括号的语句是方法调用。如

```
puts("打不开箱子")
```

puts 方法可以把字符串显示在画面上。

最后，使用 **def** 语句来定义方法。

```
def 方法名 (参数1, ...)  
  处理代码  
end
```

2.2 数据抽象和继承

多态性、数据抽象和继承被称为面向对象编程的三原则。这三项原则通常也会有别的称谓。例如，把多态性称为动态绑定，把数据抽象称为信息隐藏或封装，虽然名称不同，但是内容都是相同的。许多人认为这些原则是面向对象程序设计的重要原则¹。

¹ 三原则虽然是非常重要的，但是在面向对象编程中并不是必不可少的。比如有不支持继承的面向对象编程语言（JavaScript）和不支持封装的面向对象编程语言（CLOS, Common Lisp）。

2.2.1 面向对象的历史

新接触面向对象概念的人可能觉得它难以理解。事实上，对于从事面向对象编程有 15 年以上的我来说，有很多概念还是觉得很难理解。

自 20 世纪 60 年代末至今，面向对象的思想已经经过了 40 多年的发展。猛一看这些一步步积累起来的成果，你可能会觉得数量庞大。然而，如果沿着面向对象的发展历史一步步开始去学习的话，那么看起来很难的面向对象概念，实际上比我们想象中的要简单。

首先，我们来回顾一下面向对象的发展历史。你不必担心讲解历史过程中提到的一些陌生的词语，后面会详细说明。

Simula的“发明”

如前所述，面向对象编程思想起源于瑞典20世纪60年代后期发展起来的模拟编程语言 **Simula**。以前，表示模拟对象的数据和实际的模拟方法是互相独立的，需要分别管理，编程时需要把两者正确地结合起来，程序员的负担是很重的。因此，**Simula** 引入了数据和处理数据的方法自动结合的抽象数据类型。随后，又增加了类和继承的功能。其实在20世纪60年代后期，现代面向对象编程语言的基本特征 **Simula** 都已经具备了。

Smalltalk的发展

Simula 的面向对象编程思想被广泛传播。从 20 世纪 70 年代到 80 年代初，美国施乐公司的帕洛阿尔托研究中心开发了 **Smalltalk** 编程语言。当时的开发宗旨是“让儿童也可以使用”。在 **Lisp** 和 **LOGO** 设计思想的基础上，**Smalltalk** 又吸取了 **Simula** 的面向对象思想，且独具一格。不仅如此，它还有一个很好的图形用户界面。这个创新的语言使得世人开始了解面向对象编程的概念。

Lisp的发展

另外，位于美国东海岸的麻省理工学院及其周边地区，用 **Lisp** 语言发展了面向对象的思想。**Lisp** 和 **FORTAN**、**COBOL** 语言一样，都是最

古老的语言。与同时期登场的其他语言不同，Lisp 语言具有非常浓厚的数学背景，所以它本身具有很强的扩展功能。面向对象的特性也是 Lisp 所拥有的。因此，编程语言规格的变更、功能的扩展和实验都很容易进行，由此产生了很多创新的想法。多重继承、混合式和多重方法等，许多重要的面向对象的概念都是从 Lisp 的面向对象功能中诞生的。

和C语言的相遇

20 世纪 80 年代，世界上很多地方都在研究面向对象编程思想。AT&T 公司的贝尔实验室在 C 语言中追加了面向对象的功能，开发出了“C with Class”编程语言。开发者是 Bjarne Stroustrup，他来自距离 Simula 的起源地瑞典不远的丹麦。在英国剑桥大学的时候，Stroustrup 就使用 Simula 语言。加入贝尔实验室以后，为了能够把 C 语言的高效率和 Simula 的面向对象功能结合起来，他开发了“C with Class”编程语言。

因为当时 Simula 的处理速度是非常缓慢的，所以在他的研究领域不能使用。“C with Class”语言就演变成了后来的 C++ 语言。从这些情况来看，C++ 是直接受到了 Simula 语言的影响，而没有受到 Smalltalk 多大影响。

Java的诞生

强调与 C 语言兼容的 C++ 语言，能够写低级的方法，这是有利有弊的。为了克服低级语言的缺点，在 20 世纪 90 年代 Java 编程语言应运而生。Java 语言放弃了和 C 语言的兼容性，并增加了 Lisp 语言中一些好的功能。此外，通过 Java 虚拟机（JVM），Java 程序可以不用重新编译而在所有操作系统中运行。

现在，Java 作为在 20 世纪 90 年代诞生的最成功的语言，被全世界广泛应用。

面向对象编程方法和编程语言一样在不断地演变发展。到了 20 世纪 90 年代，面向对象的方法在软件设计和分析等软件开发的上层领域中流行起来。1994 年，当时主要的面向对象分析和设计方法 Booth、OMT（Object Modeling Technique）以及 OOSE（Object Oriented Software Engineering）的发明人 Grady Booch、Jim Rumbaugh 和 Ivar Jacobson 合作设计了 UML（Unified Modeling Language）。UML 是用

来描述通过面向对象方法设计的软件模型的图示方法，也是利用这种记法进行分析和设计的一种方法论。

UML 提供了很多设计高可靠性软件的面向对象设计方法。但是，UML 整体上很复杂，用到的概念很多，会让初学者觉得很难掌握。

面向对象的基本概念的建立，催生了各种编程语言。

2.2.2 复杂性是面向对象的敌人

我们再回到面向对象的重要原则，来了解真正的面向对象编程。

软件开发的最大敌人是复杂性。人类的大脑无法做太复杂的处理，记忆力和理解力也是有限的。

计算机上运行的软件却没有这样的限制，无论多么复杂的计算机软件，无论有多少数据，无论需要多长时间，计算机都可以处理。随着越来越多的数据要用计算机来处理，对软件的要求也越来越高，软件也变得越来越复杂。

虽然计算机的性能年年在提高，但它的处理能力终究是有限的，而人类理解力的局限性给软件生产力带来的限制则更大。在计算机性能这么高的今天，人们为了找到迅速开发大规模复杂软件的方法，哪怕牺牲一些性能也在所不惜。

2.2.3 结构化编程

最初对这种复杂的软件开发提出挑战的是“结构化编程”。结构化编程的基本思想是有序地控制流程，即把程序的执行顺序限制为顺序、分支和循环这 3 种，把共通的处理归结为例程（见图 2-7）。

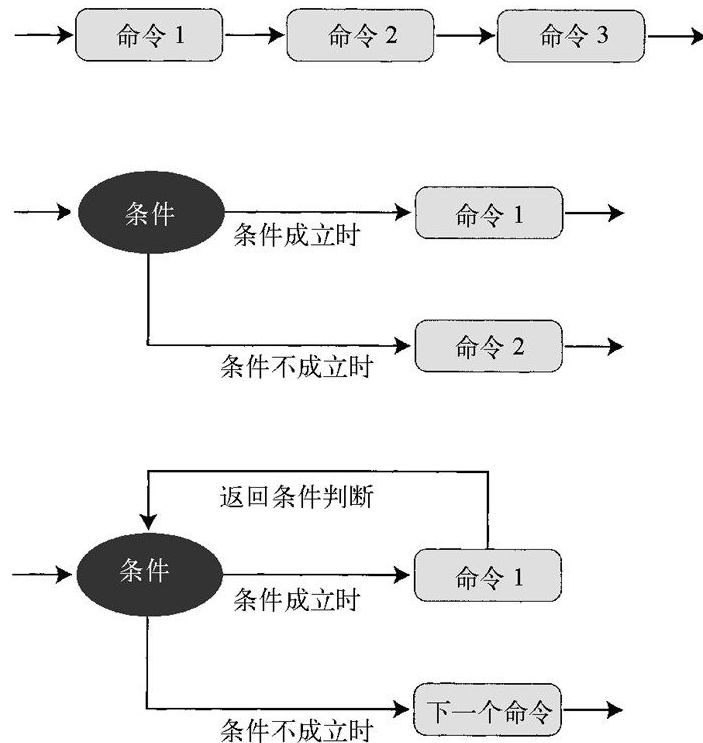


图 2-7 顺序、分支和循环的处理方法

在结构化编程出现之前，可以用 **goto** 语句来控制程序的流程，执行流可以转移到任何地方。而结构化编程用如上文所述的 3 种语句控制程序的流程。这样可以降低程序流程的复杂性，此外，还引入了较为抽象的处理块（例程）的概念，也就是把基本上相同的处理抽象成例程，其中不同的地方由外部传递进来的参数来对应。

结构化编程的“限制”和“抽象化”，是人类处理复杂软件的非常有效的方法。

通过限制大大降低了程序的自由度，减少了各种组合，使得程序不至于太过复杂。但是如果由于降低了程序的自由度而导致程序的实现能力低下，那是我们所不愿意看到的。而结构化编程的顺序、分支和循环可以实现一切算法，虽然降低了程序的复杂性和灵活性，但是程序的实现能力并没有降低。

抽象化的目的是我们只需要知道过程的名字，而并不需要知道过程的内部细节，因此它也被称为“黑盒化”。我们只需要知道“黑盒子”的输入和输出，而过程的细节是隐藏的²。

2 计算器是黑盒子的一个例子。输入数字后，计算结果在液晶屏上显示出来，而内部是怎样计算的我们并不知道。也有可能是里面的小人在打算盘哦。

例如，如果你知道了例程的输入和输出，那么即使不知道处理的内部细节也可以利用这个例程。建立一个由黑盒子组合起来的系统，复杂的结构被黑盒子隐藏起来，这样我们就可以更容易、更好地理解系统的整体结构。

如果把黑盒子内的处理也考虑上，整个系统的复杂性并没有改变。但是如果不考虑黑盒子内部的处理，系统复杂性就可以降低到人类的可控范围内。此外，黑盒子内部的处理无论怎么变化，如果输入和输出不发生变化，那么就对外部没有影响，所以这种扩展特性是我们非常希望获得的。

针对程序控制流的复杂问题，结构化编程采用了限制和抽象化的武器解决问题。结果证明，结构化程序设计是成功的，并且这种方法已经有了稳固的基础。现在几乎所有的编程语言都支持结构化编程，结构化编程已经成为了编程的基本常识。

2.2.4 数据抽象化

然而，程序里面不仅包括控制结构，还包括要处理的数据。结构化编程虽然降低了程序流程的复杂性，但是随着处理数据的增加，程序的复杂性也会上升。面向对象编程就是作为对抗数据复杂性的手段出现的。

前面已经介绍过了，世界上第一个面向对象的编程语言是 **Simula**。随着仿真处理的数据类型越来越多，分别管理程序处理内容和处理数据对象所带来的复杂性也越来越高。为了得到正确的结果，必须保持处理和数据的一致性，这在结构化编程中是非常困难的。解决这一问题的方案就是数据抽象技术。

数据抽象是数据和处理方法的结合。对数据内容的处理和操作，必须通过事先定义好的方法来进行。数据和处理方法结合起来成为了黑盒子。

举一个栈的例子。栈是先入后出的数据存储结构³。比如往快餐托盘中叠加地摞放食品（见图 2-8）。栈只有两种操作方法：入栈（push），向栈中放入数据；出栈（pop），把最后放入的数据拿出来。

3 队列是和栈相似的数据结构，是先入先出的。

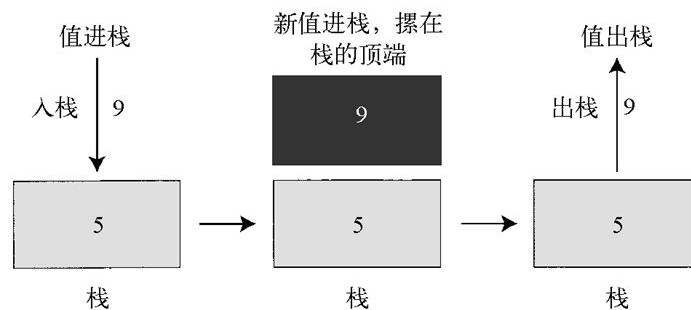


图 2-8 栈的构造

我们用 Ruby 来写这个栈⁴。图 2-9 中使用了抽象的数据结构，栈的操作只有 push 和 pop。别的方法是无法访问栈内数据的。图 2-10 中则没有使用抽象的数据结构，而是用数组索引来实现栈的操作。和图 2-9 相比，哪个更简单是显而易见的。

4 在标准的 Ruby 中没有定义栈（stack）这种数据结构。如果要执行图 2-9 所示的程序，那么图 2-11 所示的 stack 定义是必要的。图 2-9 所示只是作为一个例子来说明抽象数据的操作方法。

```
# 用 Stack.new 生成新的栈
stack = Stack.new
# 对 stack 进行 push 操作
stack.push(5)
stack.push(9)

# 用 stack 的 pop 方法取出数据
puts stack.pop() #显示 9
puts stack.pop() #显示 5
```

图 2-9 用 Ruby 写的栈的操作

```
# 用数组实现栈的操作
stack = []
```

```
# 数组的先头位置
sp= 0

stack[sp] = 5
sp += 1
stack[sp] = 9
sp += 1

sp -= 1
puts stack[sp]
sp -= 1
puts stack[sp]
```

图 2-10 用数组实现图 2-9 的程序

图 2-9 的程序有几点优于图 2-10 的程序。第一，图 2-10 的程序暴露了“数组和下标”这一内部构造，而图 2-9 则把内部构造隐藏到了 **Stack** 这一数据结构里。利用图 2-9 的方法，使用栈的人并不需要关心栈是如何实现的，即使将来因为什么事情而改变了栈的内部实现方式，也不需要在使用栈的程序做任何修改。

在这一点上，图 2-10 的程序就不一样了。如果其内部实现发生变化，也必须对自己的程序进行相应的修改。因为所有利用栈的地方都需要修改，程序规模越大，修改的工作量也就越大。所以有时候即使明知能够改善程序，也会因为工作量太大而不愿意改变栈的实现方式。“利于变化”是抽象数据的巨大优点。

另外一点是图 2-9 所示的方法很容易理解。比如数据的 **push** 操作，在图 2-9 中是：

```
stack.push(5)
```

在图 2-10 中是：

```
stack[sp]=5
sp += 1
```

图 2-9 中可以直接表现 **push** 这个操作。对数据进行操作的一方，并不需要知道图 2-10 中的处理细节，而只对“要做什么”感兴趣。所以隐藏了处理细节的程序会变得更加明确，实现目的也更清晰。

不仅是操作方法容易理解，抽象数据也是能够对特定的操作产生反应的智能数据。使用抽象数据可以更好地模拟现实世界中各种活生生的实体。

有了数据抽象，程序处理的数据就不再是单纯的数值或文字这些概念性的东西，而变成了人脑容易想象的具体事物。而代码的“抽象化”则是把想象的过程“具体化”了。这种智能数据可以模拟现实世界中的实体，因而被称作“对象”，面向对象编程也由此得名。

2.2.5 雏形

出现在程序中的对象，通常具有相同的动作。以交通仿真程序为例，程序中有表示车和信号的对象。虽然同样的对象具有相同的性质，但是位置、颜色等状态各有不同。

从抽象的原则来说，多个相同事物出现时，应该组合在一起。这就是 **DRY** 原则（即 **Don't Repeat Yourself**）。

我们已经看到，程序的重复是一切问题的根源。重复的程序在需要修改的时候，所涉及的范围就会更广，费用也就更高。当多个重复的地方都需要修改时，哪怕是漏掉其中之一，程序也将无法正常工作。所以重复降低了程序的可靠性。

进一步说，重复的程序是冗余的。人们解读程序、理解程序意图的成本也会增加。让我们再看看代码重复的图 2-10 和没有代码重复的图 2-9，显然图 2-9 的程序更容易理解。请记住，计算机是不管程序是否难以阅读，是否有重复的。然而，开发人员要阅读和理解大量的程序，所以程序的可读性直接关系到生产力。重复冗长的程序会降低生产力。复制和粘贴程序会导致重复，应该尽量避免。

让我们再回到对象的话题上。同样的对象大量存在的时候，为了避免重复，可以采用两种方法来管理对象。

一种是原型。用原始对象的副本来作为新的相同的对象。Self、Io 等编程语言采用了原型。有名的编程语言用原型的比较少，很意外的是，JavaScript 也是用的原型。

另外一种模板。比方说我们要浇注东西的时候，往模板里注入液体材料就能浇注出相同的东西。这种模板在面向对象编程语言中称为类（class）。同样类型的对象分别属于同样的类，操作方法和属性可以共享。

跟原型不同，面向对象编程语言的类和对象有明显区别，就像做点心的模具和点心有区别一样，整数的类和 1 这个对象、狗类和名字是 poochy 这条狗也都是有区别的。为了清晰地表明类和对象的不同，对象又常常被称作实例（instance）。叫法虽有不同，但实例和对象是一样的。

在 Ruby 面向对象编程语言⁵ 中，类用关键字 **class** 来声明。图 2-9 中的栈，就是 **Stack** 类。**Stack** 类的定义如图 2-11 所示。

5 Ruby 也可以用于原型（prototype）面向对象编程。

```
class Stack
  def initialize
    @stack = []
    @sp = 0
  end

  def push(value)
    @stack[@sp] = value
    @sp += 1
  end

  def pop
    return nil if @sp == 0
    @sp -= 1
    return @stack[@sp]
  end
end
```

图 2-11 Stack 类的实现

`class` 后面是类名。在图 2-11 中，`class` 后面就是 `Stack`。Ruby 规定类名称的第一个字母必须大写。类定义的最后用 `end`。在 `Stack` 这个类中，定义了 `initialize`、`push`、`pop` 这三个方法。

图 2-9 的程序第二行调用了 `initialize` 这个初始化方法。每次生成 `Stack` 对象的时候，都要调用 `initialize` 这个初始化方法。

```
stack = Stack.new
```

在图 2-11 的初始化方法中，`@stack`（实际保存栈数据的数组）和 `@sp`（数组下标）这两个变量被初始化。在 Ruby 中以“@”开头的变量用来保存每个对象中分别独立存在的值，也称为实例变量。如果你创建了多个栈对象，那么每个对象里面都分别有自己独立的 `@stack` 和 `@sp` 这两个变量。

`push` 和 `pop` 是操作栈的方法。在图 2-10 中不过是罗列了对栈的操作步骤罢了。

图 2-11 的 `initialize` 是对类定义的操作对象的内部数据进行初始化的“方法”。

为了简化说明，图 2-11 的例子中没有检查数值的范围。事实上，程序中需要检查下标是否为负值等。

2.2.6 找出相似的部分来继承

随着软件规模的扩大，用到的类的个数也随之增加，其中也会有很多性质相似的类。这就违背了我们之前强调多次的 **DRY** 原则。程序会变得重复而且不容易理解。修改程序的代价也会变高，生产力则会降低。所以，如果有把这些相似的部分汇总到一起的方法就好了。

继承就是这种方法。具体说来，继承就是在保持既有类的性质的基础上而生成新类的方法。原来的类称为父类，新生成的类称为子类。子类继承父类所有的方法，如果需要也可以增加新的方法。子类也可以根据需要重写从父类继承的方法。

图 2-12 演示了 **FixedStack** 这个类，它继承了图 2-11 中的 **Stack** 类。类名后面的“<**Stack**”指的是父类。它说明了 **FixedStack** 是 **Stack** 的子类，继承了 **Stack** 类的方法和属性。

```
class FixedStack < Stack
  def initialize(limit)
    super()
    @limit = limit
  end

  def push(val)
    if @sp >= @limit
      puts "over limit"
      return
    end
    super(val)
  end

  def top
    return @stack[-1]
  end
end
```

图 2-12 用类来继承图 2-11 中类的例子

FixedStack 类重写了 **initialize** 和 **push** 这两个方法。这两个方法都调用了 **super** 方法，这表明在子类的方法中也调用了父类的具有相同名字的方法。比如在 **FixedStack** 类的 **initialize** 方法中也调用父类 **Stack** 的 **initialize** 方法。利用这种方式，我们可以只改变子类方法的动作，而不会对父类方法产生任何影响。

initialize 方法在对象初始化时被调用。如果像下面的程序一样，在调用 **initialize** 方法时传入参数 10，那么栈对象的实例变量 **@limit** 就会被设置为 10，它是栈中元素个数的上限。

```
stack = FixedStack.new(10)
```

在图 2-12 中，程序末尾追加了并不把栈顶元素弹出栈而只是引用栈顶元素的方法 **top**。**top** 在父类中并没有定义，这是一个在子类中追加

方法的例子。

像图 2-12 这样，利用现有的类派生新类的方法称为“差分编程法”（difference programming）。通过抽象把共通的部分提取出来生成父类，与利用已有的类来生成新类，是同一方法的两种不同表现形式。前者称为自底向上法，后者称为自顶向下法。

Ruby 跟多数编程语言一样，一个子类只能有一个父类，这称为“单一继承”。从自顶向下的方法来看，通过扩展一个类来生成新的类也是很自然的。

但是，从用自底向上的方法提取共通部分的角度来看，一个子类只能有一个父类的限制是太严格了。其实，在 C++、Lisp 等编程语言中，一个子类可以有多个父类，这称为“多重继承”。

2.3 多重继承的缺点

上一节讲解了面向对象编程的三大原则（多态性、数据抽象和继承）中的继承。如前所述，人们一次能够把握并记忆的概念是有限的，为解决这一问题，就需要用到抽出类中相似部分的方法（继承）。继承是随着程序的结构化和抽象化自然进化而来的一种方式。

但最后一句话严格来说并不完全正确。结构化和抽象化，意味着把共通部分提取出来生成父类的自底向上的方法。如果继承是这样诞生的话，那么最初，有多个父类的多重继承¹就会成为主流。

1 单一继承（single inheritance）是指只能有一个父类（super class）的继承，也称为单纯继承。有多个父类的继承称为多重继承（multiple inheritance）。

但实际上，最初引入继承的 Simula 编程语言，只提供单一继承。同样，在随后的很多面向对象编程语言中也都是这样的。因此我认为，继承的原本目的实际上是逐步细化。

2.3.1 为什么需要多重继承

单一继承只能有一个父类。有时候，大家会觉得这样的制约过于严格了。在现实中，一个公司职员同时也可能是一位父亲，一个程序员同

时也可能是一位作家。

正如上一节中说明的，如果把继承作为抽离出程序的共通部分的一个抽象化手段来考虑，那么从一个类中抽象化（抽出）的部分只能有一，这个假定会给编程带来很大的限制。因此，多重继承的思想就这样产生了。单一继承和多重继承的区别仅仅是父类的数量不同。多重继承完全是单一继承的超集，可以简单地看做是单一继承的一个自然延伸（图 2-13）。

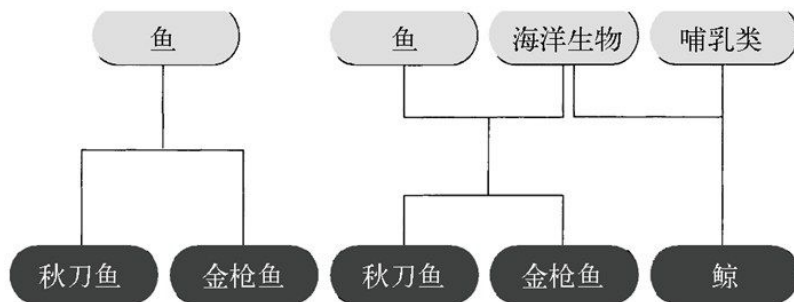


图 2-13 单一继承和多重继承的区别。在多重继承中，每个类可以有多个父类

可以使用多重继承的编程语言，不受单一继承的不自然的限制。例如，只提供单一继承的 **Smalltalk** 语言，它的类库因为单一继承而显得很自然。

Smalltalk语言中定义输入输出的**Stream** 类有3个子类。其中，**ReadStream** 是输入类，**WriteStream** 是输出类，**ReadWriteStream** 是输入输出类。**ReadWriteStream** 具有**ReadStream** 和 **WriteStream** 两个类的功能，但是由于 **Smalltalk** 是单一继承的，所以 **ReadWriteStream** 不能同时从这两个类继承。

结果是 **ReadWriteStream** 继承了 **WriteStream** 这个类，然后再把 **ReadStream** 的程序复制过来，从而实现 **ReadStream** 的功能（参见图2-14）。从程序维护的观点来看，程序复制是必须禁止的。由于单一继承的限制而导致的程序复制是我们不愿意看到的。

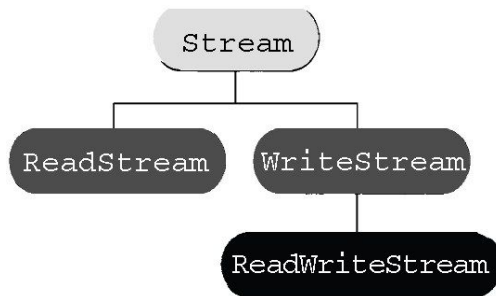


图2-14 单一继承的问题。ReadWriteStream 只能有一个父类，即 WriteStream，而不能同时继承 ReadStream

从另外的角度来看，如果有多重继承的话，那么很自然地从小 ReadStream 和 WriteStream 继承就可以生成 ReadWriteStream（参见图 2-15）。

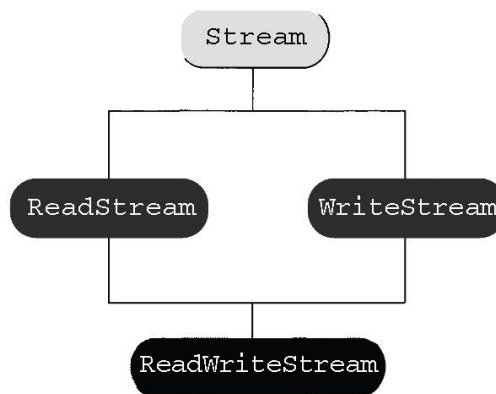


图 2-15 用多重继承的解决方法。和图 2-14 不同，ReadWriteStream 类可以继承两个父类

2.3.2 多重继承和单一继承不可分离

经过对多重继承和单一继承这样一比较，单一继承的特点就很明显了。

- 继承关系单纯

单一继承的继承关系是单纯的树结构，这样有利有弊。类之间的关系单纯就不会发生混乱，实现起来也比较简单。但是，如刚才的 Smalltalk 的 Stream 一样，不能通过继承关系来共享程序代码，导致了最后要复制程序。

对需要指定算式和变量类型的 Java 这样的静态编程语言来说，单一继承还有一个缺点，我们将在后面说明。

多重继承的特点正好相反。多重继承有以下两个优点：

- 很自然地做到了单一继承的扩展；
- 可以继承多个类的功能。

单一继承可以实现的功能，多重继承都可以实现。但是，类之间的关系会变得复杂。这是多重继承的一个缺点。

2.3.3 goto 语句和多重继承比较相似

前面我们讲到了结构化编程，说明了与其用 goto 语句在程序中跳来跳去，还不如用分支或者循环来控制程序的流程。分支和循环可以用 goto 语句来实现，单纯的分支和循环组合起来不能直接实现的控制也可以用 goto 语句来实现。goto 语句具有更强的控制力。

goto 语句的控制能力虽然很强，但是我们也不推荐使用。因为用 goto 语句的程序不是一目了然的，结构不容易理解。这样的流程复杂的程序被称为“意大利面条程序”。

多重继承也存在同样的问题。多重继承是单一继承的扩展，单一继承可以实现的功能它都可以实现。用单一继承不能实现的功能，多重继承也可以实现。

但是，如果允许从多个类继承，类的关系就会变得复杂。哪个类继承了哪个类的功能就不容易理解，出现问题时，是哪个类导致的问题也不容易判明。

这样混合起来发展的继承称为“意大利面条继承”。当然也不能说所有的多重继承都是意大利面条继承，但是使用时格外小心是必要的。多重继承会导致下列 3 个问题。

- 结构复杂化

如果是单一继承，一个类的父类是什么，父类的父类又是什么，都很明确，因为只有单一的继承关系。然而如果是多重继承的话，一个类有多个父类，这些父类又有自己的父类，那么类之间的关系就很复杂了。

- **优先顺序模糊**

具有复杂的父类的类，它们的优先关系一下子很难辨认清楚。比如图 2-16 中的层次关系，D 继承父类方法的顺序是 D、B、A、C、Object 还是 D、B、C、A、Object，或者是其他的顺序，很不明确。确定不了究竟是哪一个。相比之下，单一继承中类的优先顺序是明确了然的。

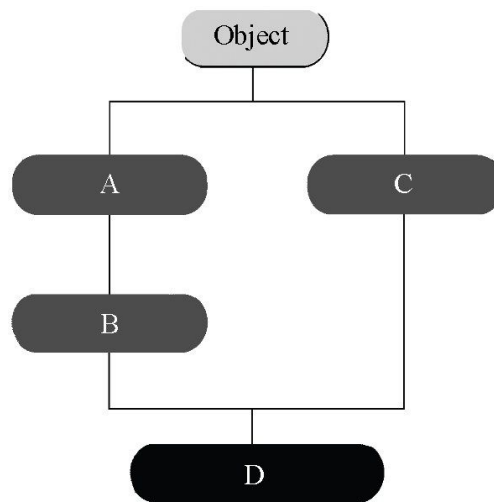


图 2-16 多重继承的优先顺序，方法调用的优先顺序不明确

- **功能冲突**

因为多重继承有多个父类，所以当不同父类中有相同的方法时就会产生冲突。比如在图 2-16 中，当类 B 和类 C 有相同的方法时，D 继承的是哪个方法就不明确了，因为存在两种可能性。

2.3.4 解决多重继承的问题

上面说明了多重继承的问题。但是像 Smalltalk 的 Stream 的例子一样，如果没有多重继承的话，有些问题还真是难以解决。

再进一步看，继承作为抽象化的手段，是需要实现多重继承功能的。在抽取类的共通功能的时候，如果一个类只允许抽出一个功能，那么限制就太多了。

既想利用多重继承的优点，又要回避它可能会带来的问题，那我们就需要寻找解决问题的方法。结构化编程解决 **goto** 问题的原则是，用 3 种有限制功能的控制语句来代替自由度太高的 **goto** 语句。这 3 种控制语句虽然有限制，但是用它们的组合可以实现任意算法。像这样引入有限制的多重继承应该是一个好的方法。

没错，受限制的多重继承，这个解决或者改善多重继承问题的方法出现了，它在 Java 编程语言中被称为接口（**interface**），在 Lisp 或者 Ruby 中是 **Mix-in**。下面我们看看这些功能是如何克服上述缺点的。

2.3.5 静态语言和动态语言的区别

我们从 Java 的接口开始说起。

在说明接口之前，首先讲一下像 Java 这样的面向对象编程语言和多重继承。

从大的方面来看，编程语言可以分为静态语言和动态语言两种。像 Java 这样规定变量和算式类型的语言称为静态语言。

在静态语言中，不能给变量赋不同类型的值，因为那样会导致编译错误。由于在编译时已经排除了类型不匹配的错误，所以在执行时就不会再发生这种错误了。不通过执行就可以发现类型不匹配这样的错误是静态语言的一个优点。

```
String str;  
  
str = "abc";    //没有问题  
str = 2;        //编译错误
```

面向对象编程语言大都用类来指定变量类型。上面例子用的就是 **String** 这个类。但是在使用面向对象编程语言时，像上面的例子那样，只能将特定类的对象（该类的实例）赋给变量的限制的确又太严

格了，因为这样的话就没有多态性了。如果只能给一个变量赋值同类对象，就不可能根据对象的类自动选择合适的处理方式（多态性）。

2.3.6 静态语言的特点

为解决这一问题，静态类型面向对象编程语言被设计成这样，当给一个类变量赋值时，既可以用这个类的对象来赋值，也可以用这个类的子类对象来赋值。这样就可以实现多态性。

请看图 2-17 中的程序。这是一个用 Java 风格的静态编程语言来定义多边形类的例子。最后出现的 **poly** 是 **Polygon** 类的一个变量，所以通过 **poly** 应该可以调用 **Polygon** 类的方法（比如“面积”方法）。但实际上，**poly** 这个变量的值是 **Polygon** 子类 **Rectangle** 的对象，所以通过 **poly** 调用的就是 **Rectangle** 的方法。当然，如果调用的方法只在 **Polygon** 中定义而没有在 **Rectangle** 中定义，那就会调用 **Polygon** 中定义的方法。

```
//多边形类
class Polygon
{
    float 面积(){...}
    int 顶点数(){...}
    ...
};

//矩形类（继承多边形类）
class Rectangle extends Polygon
{
    float 面积() {...} //再定义面积计算方法
    int 边长() {...}   //矩形类特有的方法
    ...
};

Polygon poly;
poly = new Rectangle();
```

图 2-17 父子关系的类的示例，变量不能调用子类特有的“边长”方法

但是反过来说，在程序中 **poly** 就是 **Polygon** 类的变量，即使它的值明明是 **Rectangle** 类的对象，用 **poly** 这个变量也不能调用

Rectangle 类中固有的方法（比如“边长”）。

换个说法就是，变量只是实际赋值对象的一个小观测窗口。即使作为变量值的对象有很多方法，但在使用这个变量来调用方法时，只能调用该变量类型“知道”的方法。

如果变量 **poly** 调用“边长”方法的话，静态语言就会毫不留情地报告编译错误。

而像 **Ruby** 这样没有类型定义的动态编程语言，是在程序执行时才来试着调用对象的方法，在实际对象没有可被调用的方法时程序才会报错。

2.3.7 动态语言的特点

动态语言允许调用没有继承关系的方法。比如说 **Ruby** 中定义了顺序取出某个元素的方法 **each**，数组和哈希表中都实现了这个方法。

```
obj.each {|x|  
  print x  
}
```

在静态语言中只能调用有继承关系的方法，数组、哈希表和字符串都能调用的方法，只能是在它们共同的父类（恐怕就是 **Object**）中定义。

这是单一继承的一个缺点，以后会详细说明。

在静态语言中，如果要调用类层次中平行类的方法，那么必须要有一个可以表现这些对象的类型。如果没有这个类型，可调用的方法是非常有限的。由此我们看到静态语言中某种形式的多重继承是不可少的。

2.3.8 静态语言和动态语言的比较

静态语言和动态语言各有利弊。静态语言即使不通过执行也可以检查出类型是否匹配。在一定程度上，程序的一些逻辑错误可以被自动检测出来。

但是，逐个来定义算式和变量的类型又会使程序变得冗长。只有包含继承关系的类才会具有多态性。相对于动态语言来说，静态语言就显得限制过多，灵活性差。

动态语言则正好相反。程序中有没有错误只有执行了才会知道。从可靠性来看也许会让你感觉有些不安。程序中没有类型定义，这样程序会变得很简洁，但别人看起来或许会有点难懂。

但是，只要方法名一样，这些对象都可以以相同的方式去处理。也就是说不需要深层次探索类也可以开发程序。这样生产效率就会大大提高²。

2 这种宽松的编程机制称为 Duck Typing（鸭子类型检测）。

2.3.9 继承的两种含义

像 Java 这样的静态面向对象编程语言的变量，具有限制调用方法的功能。但实际上限制的是类有什么样的方法，而不是这个类是怎么实现的。

到现在为止我们一直都在讨论继承，其实继承包含两种含义。一种是“类都有哪些方法”，也就是说这个类都支持些什么操作，即规格的继承。

另外一种是，“类中都用了什么数据结构和什么算法”，也就是实现的继承。

静态语言中，这两者的区别很重要³。Java 就对两者有很明确的区分，实现的继承用 **extends** 来继承父类，规格的继承用 **implements** 来指定接口。

3 动态编程语言中，区分规格的继承和实现的继承意义不大。即使没有继承关系，方法也可以自由地调用。

类是用来指定对象实现的，而接口只是指定对象的外观（都有哪些方法）。

Java 中，只允许用 **extends** 继承一个父类（实现的继承），所以类的继承是单一的。类的关系树和类库也就相对简单。

然而，**implements** 可以指定多个接口（规格的继承）。接口规定了要怎样处理该对象。

举个具体例子说明一下。我们来看看图 2-18 中 **java.util.Collection** 这个接口的类层次。

java.util.Collection 是定义集合的接口，有 2 个接口来继承它，分别是按顺序存放元素的 **java.util.List** 和没有重复元素的 **java.util.Set**。也就是说，实现了 **java.util.List** 或 **java.util.Set** 的对象也可以被当做 **java.util.Collection** 来处理。

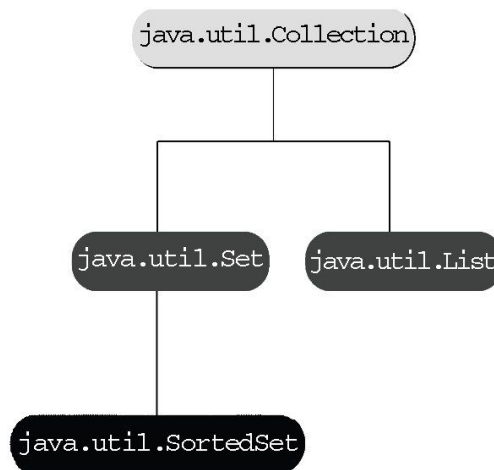


图 2-18 接口的类层次，**java.util.Collection** 的例子

接口对实现没有任何限制。也就是说，接口可以由跟实现的继承没有任何关系的类来实现。也就是说，实现这一接口的类可以继承任何其他类。例如在 **java.util** 包中，作为 **java.util.List** 的实现，既提供用数组实现的 **java.util.ArrayList**，也提供用双向链表实现的 **java.util.LinkedList**。这些类都直接继承 **Object** 类。

2.3.10 接口的缺点

关于规格继承和实现继承的区别，很久以前就有论文进行了相关的探讨。但在众多得到广泛运用的编程语言中，**Java** 是第一个实现这种功能的。这可以说是 **Java** 对多重继承问题的解答。既实现了静态语言的多重继承性，又避免了多重继承的数据构造的冲突和类层次的复杂性。

但是，我们并不能说接口是解决问题的完美方案。接口也有不能共享实现的缺点。

为了解决多重继承的问题，人们允许了规格的多重继承，但是还是不允许实现多重继承。针对这一点，我们不太好再说什么，但作为用户，就是觉得不方便。**Java** 推荐的解决共享实现问题的方案是，在单一继承的前提下，使用组合模式（**Composite**）来调用别的类实现的共通功能。

本来只是为了跨越继承层次来共享代码，现在却需要另外生成一个独立对象，而且每次方法调用都要委派给那个对象，这实在是不太合理，而且执行的效率也不高。

2.3.11 继承实现的方法

和静态语言 **Java** 不同，动态语言本来就没有继承规格这种概念。动态语言需要解决的就是实现的多重继承。

动态语言是怎么解决这一问题的呢？**Lisp**、**Perl** 和 **Python** 都提供了多重继承功能，这样就不存在单一继承的问题了。在这些语言中，使用多重继承时请千万要小心。

2.3.12 从多重继承变形而来的 **Mix-in**

Ruby 采用了和 **Java** 及其他动态语言都不同的方法。**Ruby** 用 **Mix-in** 模块来解决多重继承的问题。

Mix-in 是降低多重继承复杂性的一个技术，最初是在 **Lisp** 中开始使用的。实现 **Mix-in** 并不需要编程语言提供特别的功能。**Mix-in** 技术按照以下规则来限制多重继承。

- 通常的继承用单一继承
- 第二个以及两个以上的父类必须是 **Mix-in** 的抽象类

Mix-in 类是具有以下特征的抽象类。

- 不能单独生成实例
- 不能继承普通类

按照这个原则，类的层次具有和单一继承一样的树结构，同时又可以实现功能共享。实现功能共享的方法是把共享的功能放在 **Mix-in** 类里面，然后把 **Mix-in** 类插入到树结构里面。相对于 Java 用接口方法解决规格继承的问题，那么 **Mix-in** 可以说是解决了实现继承的问题。

我们看一个 **Mix-in** 的具体例子。针对图 2-14、图 2-15 中的 Smalltalk 的 Stream 问题，图 2-19 显示的是用 **Mix-in** 构建的一个相同结构。

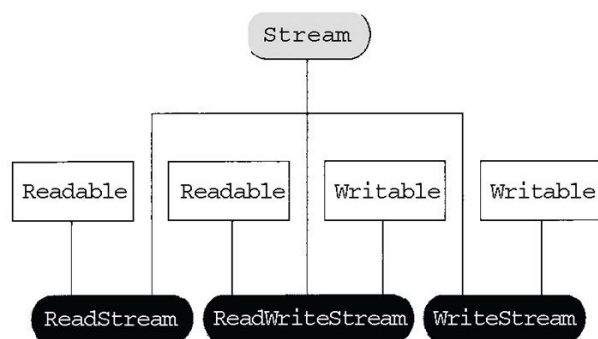


图 2-19 用 **Mix-in** 实现 **Stream** 类。既保持了类层次的树结构，又避免了复制程序

在使用 **Mix-in** 的类结构中，**Stream** 只有 3 个子类。在此基础上，实际的输入/输出处理用 **Readable**（输入）和 **Writable**（输出）这两个 **Mix-in** 类来实现。3 个子类通过继承 **Mix-in** 类而分别实现了输入、输出以及输入和输出的功能。

从 **Stream** 的类层次来看，父类是 **Stream**，负责输入输出的是 **ReadStream**、**WriteStream** 和 **ReadWriteStream** 这 3 个子类，它们形成了非常清晰的树结构。层次很简单，没有变成网状结

构。而且，由于 **Mix-in** 类实现了共通功能，从而避免了复制程序代码。

和一般的多重继承相比，**Mix-in** 是使类结构变得简单的优秀技术。使用 **Mix-in** 规则来限制多重继承，实际上也可以说是“驯服”了多重继承。

这和结构化编程用分支和循环来限制随意的 **goto** 语句是一样的。**Mix-in** 可以应用于所有多重继承编程语言中，因此，掌握这个技术是非常有必要的。

2.3.13 积极支持 **Mix-in** 的 Ruby

和其他直接引入多重继承的编程语言相比，Ruby 具有直接支持 **Mix-in** 的特点。在 Ruby 中，**Mix-in** 的单位是模块（**module**）。模块具有 **Mix-in** 的特性，即：

- 不能生成实例；
- 不能从普通类继承。

下面，我们看看在 Ruby 中是怎样使用 **Mix-in** 的。图 2-20 演示了 Ruby 是怎样实现图 2-19 的 **Stream** 类的定义的。

```
# Stream 类, Object 的子类
class Stream < Object

  # 这里的定义省略
  ...

end

# 输入用 Mix-in
module Readable

  # 定义输入用的方法
  def read
    ...
  end

end
```

```

# 输出用 Mix-in
module Writable

  # 定义输出用的方法
  def write(str)
    ...
  end

end

# 输入用Stream, Stream 的子类
class ReadStream < Stream

  # 继承输入用的Mix-in
  # Ruby 称为include
  include Readable

end

# 输出用Stream, Stream 的子类
class WriteStream < Stream

  # 继承输出用Mix-in
  include Writable

end

# 输入输出用Stream, Stream 的子类
class ReadWriteStream < Stream

  # 继承输入用Mix-in
  include Readable

  # 继承输出用Mix-in
  include Writable

end

```

图 2-20 用 Ruby 实现图 2-19 的 Stream 类的定义

模块用关键字 **module** 来定义，这和定义类用关键字 **class** 相似，但是不能指定它的父类。其中方法等的定义与类也是一样的。

在类中通过 **include** 可以继承模块中的方法。因为是继承而不是复制，所以当类中有同样的方法时，类中的方法就会被优先执行。

关于继承的各方面内容，我们都总结到了表 2-2 中。

表2-2 与继承有关的内容

用 语	内 容
单一继承	只能有一个父类，单纯但存在几个问题
多重继承	可以有多个父类，解决了单一继承的问题（面向对象的编程语言需要某种形式的多重继承），但引入了单一继承所没有的新问题
静态语言	区分规格的继承和实现的继承
动态语言	只有实现的继承
规格多重继承的问题	Java 的接口可以解决
实现多重继承的问题	Mix-in 可以解决
Mix-in	所有支持多重继承的语言都可以考虑使用
Ruby的 Mix-in	强制利用模块，积极解决多重继承的问题

2.4 两个误解

本节将说明一下关于对面向对象的误解。

作为一个很早就接触面向对象编程语言的爱好者，我写过关于面向对象的文章，开发了面向对象编程语言 Ruby。我觉得自己为让更多的人都能够熟悉面向对象编程语言作出了贡献。我骄傲地认为，Ruby 比 Smalltalk 更容易上手，比 Java 和 C++更容易实现面向对象编程，从而使人们更容易理解面向对象的概念。

但是，在这个过程中，由于我的不成熟，可能会加深一些人对面向对象的误解。

在这些误解中，有两个是我很在意的。

一个误解是，对象是对现实世界中具体物体的反映，继承是对物体分类的反映。这个观点是错误的。我之前写过的《面向对象编程语言 Ruby》¹ 中也用哺乳动物，比如狗和鲸等举过例子，可能也加深了这种误解。

1由松本行弘与石冢圭树合著，ASCII 出版社出版，ISBN 为 4756132545

(<http://www.ascil.co.jp/books/detail/4-7561/4-7561-3254-5.html>)。

另一个是，多重继承是不好的。这个观点也是错误的。这一误解好像还大都与“但 **Mix-in** 不错”² 的误解掺和在一起。

2 **Mix-in** 是 Ruby 中可以利用的一个抽象类。既具有单一继承的方法构成和优先顺序的明确性，又可以像多重继承一样从多个类继承。**Mix-in** 类不能用来生成实例，也不能继承普通类。

在解释之前我先表明一下正确的观点。关于多重继承，正确的理解应该是，如果用得不好就会出问题。对 **Mix-in** 的理解应该是，**Mix-in** 只不过是实现多重继承的一个技巧而已。

Ruby 只支持 **Mix-in** 形式的多重继承。这是因为在当时 **Mix-in** 这种技术还不广为人知，我只是想把多重继承作为一种启蒙，并没有贬低多重继承的意思。这可能造成了有人认为多重继承不好的误解。曾有一个著名的青年学者因为 Ruby 的原因而误认为多重继承和 **Mix-in** 是不同的概念。这也是我要反省的一点吧。

当然，我也不觉得这些误解全都是我造成的。但是，为了减少这些误解，下面再讲解一下面向对象编程语言和多重继承。

2.4.1 面向对象的编程

从历史上看，从 20 世纪 60 年代末期到 70 年代，分别有几个不同领域都发展了面向对象的思想。比如数据抽象的研究、人工智能领域中的知识表现（框架模型）、仿真对象的管理方法（**Simula**）、并行计算模型（**Actor**）以及在结构化编程思想影响下而产生的面向对象方法。

框架模型是现实世界的模型化。从这个角度来看，“对象是对现实世界中具体事物的反映”这个观点并没有错。

但是不管过去怎样，现在对面向对象最好的理解是，面向对象编程是结构化编程的延伸。

计算机最初出现时，对软件的要求是非常简单的，只是把人完成工作的步骤用汇编或者机器语言表现出来，编程并不是很难的工作。但是随着软件的复杂化，开发就变得越来越复杂。因为这个原因，Edsger Dijkstra³ 提倡把程序控制限制为以下 3 种的组合，使程序变得简单且容易理解。

3 Edsger Wybe Dijkstra 是荷兰的计算机科学家。他提倡结构化编程，发起了减少使用 goto 语法的运动。他也是解决图论中最短路径问题的 Dijkstra 方法的研究者。

1. 顺序——程序按照顺序执行。
2. 循环——一定的条件成立时程序反复执行。
3. 分支——条件满足时执行 A 处理，不满足时执行 B 处理。

结构化编程基本上实现了控制流程的结构化。但是程序流程虽然结构化了，要处理的数据却并没有被结构化⁴。面向对象的设计方法是在结构化编程对控制流程实现了结构化后，又加上了对数据的结构化。

4 20 世纪 70 年代，Michael A. Jackson 在开发的 Jackson Structured Programming (JSP) 中尝试了数据的结构化。但是，JSP 中的结构化对象是操作数据的流程而不是数据。

众多面向对象的编程思想虽不尽一致，但是无论哪种面向对象编程语言都具有以下的共通功能。

1. 不需要知道内部的详细处理就可以进行操作（封装、数据抽象）。
2. 根据不同的数据类型自动选择适当的方法（多态性）。

可以这样认为，以上两点在面向对象编程语言中是必不可少的。因为不必知道内部结构，所以可以把数据当做黑盒来操作。即使将来数据结构发生变化，对外部也没有影响。黑盒化是模块化的基本原则，面向对象编程语言将每一类数据都当做黑盒处理。

多态性是根据不同的数据类型而自动选择适当的处理。这就不需要由人来根据不同的数据类型对处理进行分支了。如果没有多态性，那么

程序中就会到处都是分支处理。这也就意味着，变更和追加数据类型会变得非常困难。

在面向对象分析和面向对象设计领域，有些观点还不尽一致，但如果只谈面向对象编程，就可以认为封装和多态是提高生产率的技术。

结构化编程通过整理数据流，提高了程序的生产效率和可维护性。同样，面向对象编程通过对数据结构的整理，提高了程序的生产效率和可维护性。

如果把面向对象编程看做是对结构化编程的扩展，那么对象是否是现实世界中具体物体的反映就不重要了。实际上，面向对象编程语言中的对象，像字符串、数组和范围等，很多都没有现实世界中的具体物体与之对应。即使现实世界中有具体物体与之对应，对象也只是描述现实物体某一侧面的抽象概念而已。比如猫有颜色、血统等很多属性，而程序中的对象并不需要把这些属性都考虑进去。程序只是处理抽象数据的。

2.4.2 对象的模板 = 类

很多面向对象编程语言都具有类和继承这两个基本特性。利用这两个特性，我们可以高效地把抽象的数据通过类封装起来。

类是对象的模板，相当于对象的雏形。在具有类功能的面向对象编程语言⁵中，对象都是由作为雏形的类来生成的，对象的性质也是由类来决定的。通过类可以把同一类的对象管理起来。

5 在有些编程语言中，类并不是必需的。相对于基于类的面向对象语言来说，那些类不是必需的面向对象语言称为基于原型的语言。有代表性的基于原型的语言有 Self。Ajax 背后的 JavaScript 也是基于原型的语言。

图 2-21 显示的是 Ruby 中对类的定义。我一般情况下是用阿猫阿狗来举例的，但为了避免误解，这次用数据结构的栈来举例。

```
class Stack

  def initialize
    @data = []
  end
end
```

```
end
def push(x)
  @data.push(x)
end
def pop
  @data.pop
end
end

s1 = Stack.new
s1.push(1)
s1.push(2)
puts s1.pop      # 显示2

s2 = Stack.new
s2.push("foo")
puts s2.pop      # 显示foo

puts s1.pop      # 显示1
```

图 2-21 类具有把相同种类的对象进行统一管理的功能。往栈 **s1** 里放入 1、2 之后，从中取出一个元素。往栈 **s2** 里放入 **foo** 之后，再从中取出一个元素，最后从 **s1** 中再取出一个元素。这就是 Ruby 程序

栈是 **LIFO**（后入先出）的数据结构。可以按照从后向前的顺序把里面的数据取出来。图 2-21 的程序里定义了两个栈，**s1** 和 **s2**，分别对它们放入和取出数据。两个栈都是由相同的类生成的对象，操作方法都一样。但是，它们的数据都是独立的，交互对两个栈进行操作也不会破坏彼此的数据。

我们可以把面向对象编程语言的类看做是结构化编程语言的结构体或记录的扩展。不同的是，类里面不仅有被称为成员或字段的数据，而且还有对这些“数据块”进行操作的方法。

对于只是把数据组织在一起的结构体，我们能做的只是取出或者更新成员变量的值。而类中定义有成员函数（也称为方法），可以调用这些方法来处理类的对象。

例程能够把一系列的处理步骤组织在一起，把处理的内容黑盒化，是个很有用的工具，而类则是把数据黑盒化的工具。由于对类内部数据的操作都是通过类的方法来实现的，所以内部数据结构即使在以后发

生变化，对外部也没有影响。这和例程把处理黑盒化之后，内部算法变化对外部没有影响是同样的道理。

像这样不用考虑内部处理的黑盒化也被称为抽象化，是降低程序复杂度的有效方法。

2.4.3 利用模块的手段 = 继承

类以数据为核心，把与之相关的处理也都集中到一起。这样，模块之间一些具有共通性质的内容就会重复出现，从而违背了禁止重复的 DRY 原则⁶。

⁶ DRY (Don't Repeat Yourself) 原则就是彻底避免重复。这一原则对提高程序开发的效率和可靠性非常有效。

避免重复的方法是继承。那些具有相同性质的类可以从拥有共通性质的类中“继承”这些共通的部分。不单是可以继承，还可以替换，追加其中不同的部分，从而生成新的类。

从这个角度来看，类是模块，继承就是利用模块的方法。继承的思想好像有其现实的知识基础，但是把它看做纯粹的模块利用方法则更恰当。

因为继承只不过是抽象的功能利用方法，所以不必把对继承的理解束缚在“继承是对现实事物的分类的反映”。实际上这样的想法反而妨碍了我们对继承的理解。

关于继承，规格继承和实现继承的区别也是非常重要的话题。规格就是从外部看到的类的功能，这样的继承是规格继承。实现继承是指继承功能的实现方法。

传统面向对象编程语言是一下子把规格和实现都继承下来，在最近的编程语言中，有的是把这两种继承分开了。比如 Java 里的接口就是规格继承，而在 Sather 编程语言中，规格继承和实现继承被完全分离开了。

2.4.4 多重继承不好吗

在早期的面向对象编程语言中，其功能被继承的类（基类或者父类）被限定为一个，这称为单一继承（或者单纯继承）。

把单一继承自然地扩展，一个类可以继承多个类的功能，就成为了多重继承。在自然界中也是一样，家长不只有一个。一个程序员同时也可能是一个父亲，同时具有多个角色。所以从单一继承到多重继承是很自然的。

但是，像 **Java** 和 **Smalltalk** 这样，不支持多重继承的编程语言还有很多，在程序员中多重继承好像也不是很普及，其中认为“多重继承是不好的东西”的人并不少。

单一继承的类之间的关系是很单纯的树结构。但是对多重继承而言，类之间的关系却是复杂的网状结构。

正因为如此，多重继承在一部分开发者当中的评价并不好，但是考虑到程序的生产力，多重继承还是必要的。

对于像 **Java** 或者 **C++** 这样需要指定变量类型的静态语言来说，父类类型的变量可以用子类的对象来赋值。如果用子类以外的对象来赋值的话，就会发生编译错误。所以可以说这既实现了多态性，又实现了对变量类型的检查，是一个很好的想法⁷。

⁷ 子类对象拥有父类所有属性，可以当做父类对象来处理，这种状态称为 **LSP**（**Liscov Substitution Principle**）。

结果是，静态语言中可以实现多态性的只是限于拥有共通父类的对象。

但是，把对象统一处理的观点可能不止一个。比如对于字符串类，如果着眼于能够比较大小这一性质的话，我们有时想把它与数值等统一处理。这时我们可能会创建一个能够比较大小的父类，让数值和字符串来继承这个父类。

而在同一程序中别的地方，考虑到字符串类是字符的序列，为能实现把其中的元素按照顺序取出的操作，又想把它与列表等统一处理。这就需要给字符串和列表定义一个共通的父类。但是单一继承只能有一个父类，不可能同时实现比较大小和按顺序访问这两个要求。

所以，为了解决这个问题，多重继承在静态编程语言中是必要的。实际上，静态面向对象编程语言的代表 C++ 和 Eiffel⁸ 都支持多重继承。Java 也可以通过接口来支持规格的多重继承。

⁸ Eiffel 是在 20 世纪 80 年代后期由 Bertrand Meyer 设计的面向对象编程语言。其主要特点是静态类型与多重继承，严密的规格与“基于契约的设计”。Eiffel 在国外金融等领域中得到了实际应用。

2.4.5 动态编程语言也需要多重继承

动态编程语言没有类型检查，从这方面来说没有理由用多重继承。那么动态编程语言真的不需要多重继承吗？

肯定不是这样的。

无论从类型上考虑结果如何，从模块的角度来看，单一继承也有很多不便性。比如“一个文件只能利用一个库”这样的限制就让我们感到很不自由。

当然，实现的共享可以通过多个对象的组合（composition）和委托（delegate）⁹ 来做到，Java 中就推荐这种方法。

⁹ 组合是把多个对象合成一个对象来处理。委托是把对一个对象的方法调用委派给别的对象。

但是如果把类当做模块来看的话，多重继承相当于语言功能支持模块组合。有了多重继承，同样的处理可以简单地记述，可以促进实现的共享。从 DRY 原则的角度来看，今后的面向对象语言也应该支持多重继承。

2.4.6 驯服多重继承的方法

多重继承因为有多个父类，所以可能引发下面两个问题。

1. 类关系复杂化。
2. 继承功能名字重复。

最初的问题起因是类的关系从简单的树结构变成了复杂的网状结构。单一继承时，子类和父类、父类和它的父类.....之间的关系是一条直线。

多重继承时，类之间的关系变成由一个类作为顶点的有向图。如图 2-22 所示，优先级不能被简单地确定。图 2-22 左边显示的是单一继承的例子。类 C 和父类之间的优先级是 C-B-A，简单而明确。右边显示的是多重继承的例子。类 5 的父类有类 1、类 2、类 3 和类 4，但是父类之间的优先级并不明确。

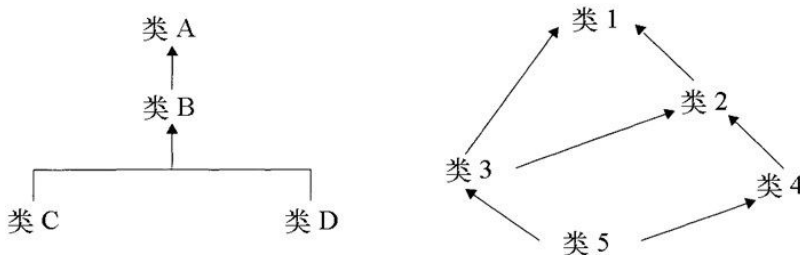


图 2-22 单一继承和多重继承

解决优先级问题需要巧妙的设计，设计好的话，就不会有（或难以发生）问题。多重继承确实容易让类之间的关系变得复杂。不管怎么说，和单一继承相比，这是一个很显眼的缺点。但是如果能够进行巧妙和适当的设计，大部分场合这个问题是可以避免的。

多重继承设计的一个有效的技巧是 **Mix-in**。Ruby 也利用了这个技巧。

用 **Mix-in** 做多重继承设计时，从第 2 个父类开始的类要满足以下条件。

1. 不能单独生成实例的抽象类。
2. 不能继承 **Mix-in** 以外的类。

满足这两个条件的类称为 **Mix-in** 类。正是因为这些限制，**Mix-in** 类可以说是功能模块。通过 **Mix-in** 类的功能和一般类的组合，继承关系既单纯，又可以享受多重继承的优点。

有这么多的限制，对于 **Mix-in** 的实用性，恐怕有人会抱有怀疑的态度。

其实一般的继承是可以变换成基于 **Mix-in** 的关系的。请看图 2-23 和图 2-24。图 2-23 是 **Window** 类的多重继承关系。在一般的 **Window** 类上，加上标题栏就是 **TitledWindow** 类，加上边框就是 **FramedWindow** 类，既有标题又有边框的是 **TitledFramedWindow** 类。**TitledFramedWindow** 类分别继承了 **TitledWindow** 类和 **FramedWindow** 类。

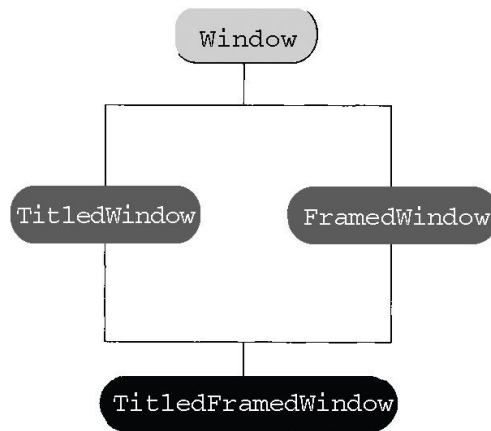


图 2-23 多重继承的例子，**Window** 类的继承关系

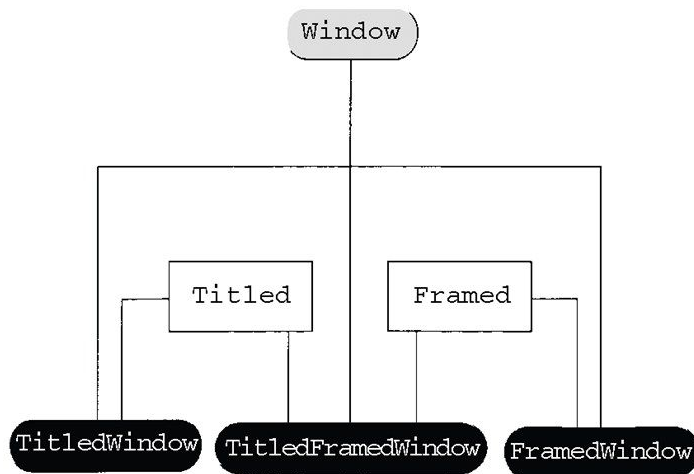


图 2-24 图 2-23 的 **Mix-in** 版

而在图 2-24 中，用 **Mix-in** 实现了相同的功能。标题功能和边框功能分别被做成两个 **Mix-in** 类，这样 **TitledWindow** 类、

FramedWindow 类和 **TitledFramedWindow** 类就成为了 3 个独立的类。

这样，通过对功能的分离，多重继承就可以由单一继承加上 **Mix-in** 类来实现。利用 **Mix-in** 就可以同时享有单一继承的单纯性和多重继承的共有性。

另外一个比较麻烦的问题是名字重复。多重继承编程语言都有自己的对应方法，大致上分为以下 3 种。

1. 给父类定义优先级

重复的时候使用优先级高的父类属性。**Common Lisp Object System (CLOS)** 提供的这个功能在继承数据类型时很有效。

2. 把重复的名字替换掉

Eiffel 使用的就是这种方法。在模块继承时用这种方法很有效，其缺点是写程序时很复杂。

3. 指定使用类的名字

C++用的是这种方法。这也是在继承模块时有效的方法。缺点是本来不需要指定类名的情况现在却要指定。

从对应方法可以明白地看到各种语言的特点。**Lisp** 重视数据类型的继承，**Eiffel** 和 **C++**重视模块的继承。

2.4.7 Ruby中多重继承的实现方法

当初设计 **Ruby** 的时候，**Mix-in** 并不广为人知。我认为 **Mix-in** 是解决多重继承问题的非常好的方法。所以，出于启蒙的目的，我特意在 **Ruby** 中强制采用了 **Mix-in**，而没有使用普通的多重继承。

不知道是否因为这个原因，**Mix-in** 的知名度提高了。但是也像前面说过的一样，甚至一些计算机科学的研究者也产生了对多重继承的误解。

我们比较一下在 Ruby 中使用和不使用 **Mix-in** 的区别。图 2-25 是使用 **Mix-in** 的 Ruby 程序。用 **module** 定义的是 **Mix-in** 类。

图 2-25 的 **LockingMixin** 可以对任意的类提供 **lock** 功能。在这里，给 **Printer** 类增加了 **lock** 功能。在 **spool** 方法中调用了 **lock** 方法。

```
module LockingMixin
  def lock
    ...
  end
  def unlock
    ...
  end
end

class Printer<Device
  include LockingMixin
  def spool(text)
    lock
    ...
    unlock
  end
end
```

图 2-25 利用 **Mix-in** 的 Ruby 程序

图 2-26 是没有 **Mix-in** 的 Ruby 程序。这里增加了实现 **Lock** 功能的对象初始化，添加了 **Lock** 方法，还要定义很多方法的委托调用。比较起来，**Mix-in** 程序就很简洁。

```
class Lock
  def lock
    ...
  end
  def unlock
    ...
  end
end
```

```

class Printer<Device
  def initialize
    @lock = Lock.new
  end
  def lock
    @lock.lock
  end
  def unlock
    @lock.unlock
  end
  def spool(text)
    @lock.lock
    ...
    @lock.unlock
  end
end
end

```

图 2-26 不用 Mix-in 来实现图 2-25 的功能

2.4.8 Java实现多重继承的方法

Java 采用了单一继承。但是为了满足静态编程语言对多重继承的需要，Java 采用了规格的多重继承，即接口。如果使用接口，即使对没有继承关系的不同种类的对象也可以做共通的处理。

但是接口只能实现规格的多重继承，实现的多重继承在 Java 中是不允许的。这种设计原则多少会让人感觉到不方便吧。

因为不允许实现的多重继承，如果要共通实现的话，一般要像图 2-26 所示的程序一样使用委托的方法。图 2-27 是使用委托来共通实现的 Java 程序例子。它把从接口调用的方法，都明确地委托给实现共通功能的对象。本来在多重继承中可以自动实现的，现在要通过手工来实现。虽然有些麻烦，但这也算 Java 的风格吧。

```

interface LockingMixin {
  void lock();
  void unlock();
}

class Lock {
  void lock(){...};
  void unlock(){...};
}

```

```

}

class Printer implements LockingMixin {
    final Lock lock = new Lock();
    void lock() {lock.lock();}
    void unlock() {lock.unlock();}
    void spool(TextData text){
        this.lock();
        ...
        this.unlock();
    }
}

```

图 2-27 用委托来实现多重继承的 Java 程序

图 2-28 没有用委托的方法，而是把实现共通功能的对象作为成员变量来使用。这样操作对象并不需要直接实现接口，而只是作为属性保存一个实现共通功能的对象，在程序中直接调用该属性的方法。

```

interface LockingMixin {
    void lock();
    void unlock();
}

class Lock implements LockingMixin {
    void lock(){};
    void unlock(){};
}

class Printer{
    final Lock lock = new Lock();
    void spool(TextData text){
        this.lock.lock();
        ...
        this.lock.unlock();
    }
}

```

图 2-28 不用委托来实现多重继承的 Java 程序

没有了委托的方法，这些部分就变得简单明了，但是在调用共通功能的时候，每次都要引用属性加上`.lock`，会让人觉得不怎么漂亮。

* * *

本节回顾了多重继承，要点有以下 5 个。

1. 多重继承并不可怕。
2. 今后面向对象编程语言必须有某种形式的多重继承。
3. 类既有类型的一面又有模块的一面。
4. C++、Eiffel 等语言积极利用了类的模块的一面。
5. 使用 **Mix-in** 可以避免多重继承的类关系变复杂。

正确地使用多重继承是提高程序效率的有效方法。如果本节的说明能够减少对多重继承的误解，那我就感到很幸运了¹⁰。

¹⁰ 本节内容参考了《面向对象入门》一书，该书由 Bertrand Meyer 著，二木厚吉审校，酒匂宽・酒匂顺子译，Ascii 出版，ISBN4756100503。书名是“入门”，但根本不是面向初学者的，而是面向中高级读者的书。例题都是用（大家不太熟悉的）Eiffel 语言写成的，遗憾的是 Eiffel 本身的版本也很老，但即使有这些缺点，这本书还是非常有价值的。例题以外的内容一点也不过时。如果想要进一步深入理解面向对象编程的话，这本书可以说是最好的。还有，翔泳社重新翻译了这本书，分两册出版：《面向对象入门（第 2 版）：原则・概念》，《面向对象入门（第 2 版）：方法论・实践》。

2.5 Duck Typing 诞生之前

在编程世界中，经常提到静态（static）与动态（dynamic）这样的词汇。静态是指程序执行之前，从代码中就可以知道一切。程序静态的部分包括变量、方法的名称和类型以及控制程序的结构等等。

相对于静态，动态是指在程序执行之前有些地方是不知道的。程序动态的部分包括变量的值、执行时间和使用的内存等等。

如果知道程序使用的算法和输入值，虽然有时候不执行也可以知道输出的结果，但是现实中这种单纯的情况很少。通常情况下，程序本来就是不被执行就不知道结果的，所以从一定程度上说程序都具有动态特性。因此，严格地说，静态和动态之间的界限是很微妙的。

2.5.1 为什么需要类型

在程序中具有动态或静态特性的东西很多，这里以类型为重点，讲解一下静态类型和动态类型。

编程语言中的类型指的是数据的种类。例如整数和字符串都是数据的类型。从硬件的角度来看，计算机可以处理的类型只有二进制。在计算机可以直接操作的汇编语言中，数据类型都是整数¹，其他类型的数据都用整数来表现。

¹ 只处理二进制数的说法只是一个概念。实际上 CPU 可以直接处理浮点小数等整数以外的类型。

例如表现字符串的时候，是给每个字符都编上号，这些整数编号排列起来构成了字符串。内存中的地址（位置）也是用整数来表现的。数组、对象等复杂数据也是一样的。

但是这种处理方法是很低级的，它要求人要理解、记忆用整数来表达所有类型数据的方法。不小心出一点差错程序就不能运行。

这样的话程序员的负担就太大了，所以编程语言就进化了。被称为世界上最初的编程语言的 FORTRAN（Formula Translator，公式变换机），引入了变量和算式的类型。在程序中，变量只能用整数赋值，数组只能是浮点数的数组等，可以指定数据类型。这是静态数据类型的开始。这种对数据类型的定义称为类型定义。图 2-29 是 C 语言的类型定义，它是静态语言的代表。

```
int i;    /*i 是整数*/  
float f;  /*f 是浮点小数*/  
char *s;  /*s 是字符串*/  
int j;    /*j 是整数*/  
  
j = "a";  /*类型不匹配*/
```

图 2-29 C 语言的类型定义

假设在程序中把字符串赋值给整数型的变量，那么根据程序的定义，编译器知道赋值语句中值的类型（字符串）和变量的类型（整数），

所以能够检查出这种类型不匹配的错误。静态的类型不用执行程序就可以通过机器检测到这种人为错误，可以说是一项伟大的发明。

2.5.2 动态的类型是从Lisp中诞生的

在 FORTRAN 出现数年之后诞生的 Lisp 编程语言（List Processor，列表处理机），对于数据类型的问题采取了另一种解决方法。在 1958 年刚出现时，Lisp 只支持列表（list）和原子（atom）这两个数据类型。

列表是可以由两个引用的节点（node）构成的单向列表，比如像(5 13)这样的数据。原子比较难说明，简单地说就是指 list 以外的数据类型，比如数值和字符串都是原子（参见图 2-30）。

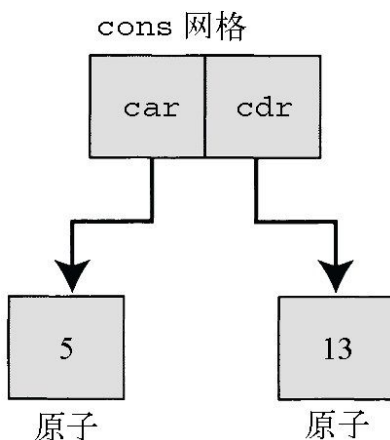


图 2-30 Lisp 的链表与原子

List 的每个节点，历史上称为 **cons** 单元，可以引用其他的 **cons** 单元或原子。**cons** 单元可以有两个引用，因为历史的原因，前面的称为 **car**，后面的称为 **cdr**（参见图 2-31）。

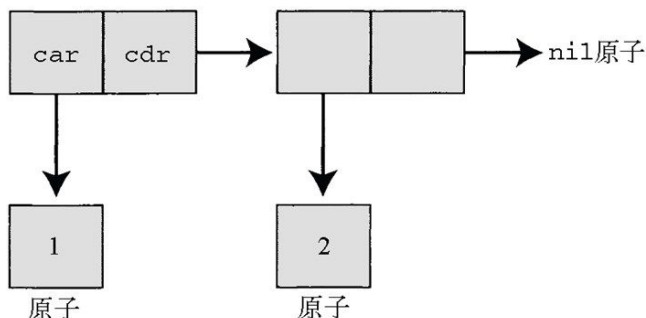


图 2-31 链表原子的细节，nil 是空的原子

在 **Lisp** 中程序和数据如果不能用文字来表现的话会很麻烦。所以 **Lisp** 用字符串来表现名为 **S** 式的列表。**S** 式是用以下规则把列表转换成字符串的。

- **cons** 单元中，**car** 的值和 **cdr** 的值用点连接，再用括号括起来。
- **cdr** 如果是列表的话，省略括号。
- 末尾的 **cdr** 如果是 **nil**，那么省略 **.nil**。

按顺序应用这些规则的话，图 2-31 的列表可以用下面的字符串来表示。从第一行开始按照顺序进行简化，就可以得到第三行的结果。

```
(1 . (2 . nil))  
(1 2 . nil)  
(1 2)
```

Lisp 的数据用列表，程序也用列表，所有的东西都用列表来表示。**Lisp** 用 **S** 式表示的列表构造本身就是程序。图 2-32 是 **Lisp** 的计算阶乘的程序。

```
(defun fact (n)  
  (if (= n 0)  
      1  
      (* n (fact (- n 1)))))
```

图 2-32 **Lisp** 的计算阶乘的程序

Lisp 的列表中的各个 **cons** 单元是指向 **cons** 单元还是原子，事前是不知道的。列表中 **cons** 单元和原子混杂存在，这从本质上就可以说是多态的数据结构。以这种数据结构为基础的 **Lisp** 采取的战略是数据要记录与自己数据类型有关的信息。这样的数据类型称为动态类型。

Lisp 程序中，如果用只能处理原子数据的方法来处理 **cons** 单元数据，执行时的数据类型检查就会报告错误。因为执行时有类型检查，

所以一旦发现有不正确的处理，程序就会停止执行。但是不执行程序的话是无法知道哪儿有错误的。

2.5.3 动态类型在面向对象中发展起来了

编程语言的数据类型分为两类，一类是起源于 FORTRAN 的指定了变量或算式数据类型的静态类型，另一类是起源于 Lisp 的动态类型。静态类型从 FORTRAN 开始，通过 COBOL、ALGOL 被很多编程语言采用。

C 语言的原型是没有数据类型定义的 BCPL 语言，该语言受到了 ALGOL 语言的影响采用了静态类型。

在很长的时间里，只有 Lisp 和受其影响的语言（比如 LOGO）才采用了动态类型，但是以“某件事情”为契机，动态类型开始得到广泛接受。

那个“某件事情”，就是面向对象编程。最初的面向对象编程语言 Simula，受到了 ALGOL 语言的很大影响，像整数等这些基本的数据类型都采用了静态类型。但是对新引入的对象，不论它是哪个类的对象，全都用 Ref 这种类型来表现。

不管是什么类的对象，它们的静态类型都是 Ref 型，没有任何区别。但是 Ref 型数据知道自己是什么类的对象，所以 Ref 可以说是动态类型。

仔细想想，对象保存着有关自己种类的信息，某个变量可以用各种类型的数据来赋值，这两点是多态这一面向对象重要特性的必要条件。因为如果变量类型和赋值数据的类型必须是完全一致的静态类型的话，程序执行时就不可能根据数据类型的不同来自动选择合适的处理方法。如果没有从 Lisp 中起源的动态类型，面向对象可能也不会存在吧。

继承了起源于 Simula 的面向对象思想，Smalltalk 像 Lisp 一样，全面采用了动态类型。虽然从外部来看，Smalltalk 和 Lisp 完全不一样，但从内部构造来看，它们像双胞胎一样。另外，Lisp 也增加了面向对象的功能，独立地发展到了现在。

无论如何，从 20 世纪 70 年代到 80 年代，面向对象编程是由动态类型语言 Smalltalk 和 Lisp（及它们的各种方言）所支撑的。

2.5.4 动态类型和静态类型的邂逅

20 世纪 80 年代，面向对象编程语言的主流是包含动态数据类型的语言。但是在 21 世纪的今天，使用最广泛的面向对象编程语言是具有静态数据类型的 Java 和 C++。在面向对象编程语言的历史上，究竟发生了什么呢？

在 20 世纪 80 年代初期，受到 Simula 的影响，一个面向对象的编程语言诞生了，这就是 C++。C++ 包含了 C 的所有功能，并增加了由 Simula 发展而来的面向对象功能²。

2 C++ 没有受到 Smalltalk 的影响，而是受到 Simula 的直接影响，所以很多用语都不同于 Smalltalk。比如，父类称为基类（base class），子类称为派生类（derived class）。这反映了 C++ 的作者 Bjarne Stroustrup 曾是 Simula 用户。

在 Simula 中，除对象以外的数据类型都是静态类型。受到 Simula 很大影响的 C++，因为引入了一个原则，对象也采用了静态类型。

这个原则就是子类对象可以看成是父类对象。具体来说，如果 String（字符串）类是 Object（对象）类的子类的话，那么 String 类的所有对象都可以看做是 Object 类的对象（参见图 2-33）。

```
class Object {
    ...;
};
class String : public Object {
    ...;
};
//看成是Object
String *str = new String();
Object *obj = str;
```

图 2-33 C++对象与静态类型的关系

根据这个原则，在编译时就可以知道变量或算式的类型，又可以根据执行时的数据类型自动选择合适的处理，从而同时具备了静态类型的优点和动态类型的多态性。

因为在编译时就知道了变量和算式类型，所以可以在执行前就发现类型不匹配错误。这是一个很大的优点。另外，根据类型信息在编译时大胆进行优化，可以提高程序执行速度。

因为有这样的优点，C++与 20 世纪 90 年代受到 C++影响诞生的 Java，以及 C#等采用静态数据类型的面向对象编程语言，都得到了广泛的使用。

2.5.5 静态类型的优点

现在静态类型的面向对象编程语言被广泛使用。首先，我们比较一下静态类型和动态类型的优缺点。

静态类型最大的优点是在编译时能够发现类型不匹配的错误。当然，在编译中是不可能发现程序中所有问题的，但是由于很多问题都是由类型不匹配引发的，所以虽然不能发现全部问题，但这种自动发现问题的功能对我们的帮助还是很大的。

与其相反，动态类型的编程语言至多只能发现程序语法错误。

程序中如果明确指定了数据类型，那么编译时可以用到的信息就很多。利用这种信息可以在编译时对程序做优化，提高程序执行速度。

数据类型信息不只是对编译器有用。我们在看程序的时候，“这个参数是什么类型”的信息对我们理解程序也是有很大帮助的。集成开发环境（IDE）也可以利用这些信息来自动补充完整方法名。这些功能的实现都得益于可以利用的类型信息。

最后，变量和算式分别有自己的类型，这使得我们能够在一开始就认真考虑这些变量应该扮演什么样的角色。我们在编写程序时就要考虑数据类型，虽然要考虑的东西变多了，但是也不能简单地说这是坏事。显而易见，这是我们开发好的、可靠性高的程序所必需的。

从以上几点来看，静态类型似乎全都是优点。其实它也有几个缺点，或者说是问题。

其中一个问题是，若不指定类型就写不了程序。当然，指定类型是静态类型编程语言的特征之一。但是说到底，数据类型只是一些辅助信息，并不是程序本质。当我们想把精力集中到程序处理的实际问题时，却要一个个考虑数据类型的定义，这是很烦琐的。并且，有时会让人觉得，有的类型声明仅仅是为了满足编译器的要求。程序规模也因为数据类型的定义而变大，重要的部分反而容易被忽视。

另外一个问题是灵活性的问题。静态类型本身限制了给某个变量只能赋值某种类型的对象，这种限制可能成为妨碍将来变化的枷锁。前面学过的多重继承和接口会产生令人费解的继承关系，这时怎样设定适当的类型就变得比较困难了。

总结一下，用静态类型编程语言的人通过定义类型，把更多的信息传达了出来，这算是给编译器和将来读程序的人减轻负担的一种方法吧。

2.5.6 动态类型的优点

前面介绍了静态类型，那么动态类型又怎样呢？动态类型编程语言的最大优点是源代码变得很简洁。编程语言的进化使我们可以用更简单的程序来传达给计算机更多信息。如果不用指定与程序本质无关的数据类型，程序也完全可以正确执行，也可以检测出来错误的话，这不是一种很好的想法吗？

得益于简洁，我们在编写程序的时候就不用考虑数据类型这些无关本质的部分了，而是可以集中于程序处理的本质部分，编写简洁程序的话，也可以提高生产力。

另一方面，有人会担心，简洁的程序虽然让我们在编程的时候变得简单了，但是因为没有类型信息，以后读起来是不是就变得难以理解呢？虽然写的时候容易了，但难读的程序也是不可取的。对于这样的担心，我的回答是，简洁的程序更突出了程序处理的实质，理解起来反而变得简单了。实际上，动态类型的编程语言（例如 **Ruby**）的程序规模和静态类型相比，程序行数相差数倍的情况并不少见。很多人都感觉到动态类型的程序更好理解。

对于简洁程序的另外一个担心是，动态类型语言是否运行缓慢呢？事实上是这样的。同样的处理，在大多数情况下，静态类型编程语言运行得要快些。

这是因为动态类型程序执行时要做类型检查。另外，静态类型的编程语言大都通过编译把程序源代码转换成可以直接执行的形式，而动态类型的编程语言大多是边解释源代码（转换成内部形式）边执行，这种编译型处理和解释型处理的区别也是影响程序执行速度的原因之一。说起程序，很多时候执行速度并不是很重要的，随着计算机性能的提高，执行速度就更不是什么严重的问题了。

动态类型编程语言的另外一个特点是灵活性。动态类型语言的程序不用指定变量的数据类型，所以即使开发时没有考虑到的数据类型也可以轻松地处理。这种灵活性的关键是我们下面要讲的 **Duck Typing** 概念。

动态类型编程语言的最大缺点是不执行就检测不出错误。和静态类型的自动错误检测相比，这算是它的不足吧。

2.5.7 只关心行为的Duck Typing

表达动态类型灵活性的概念是 **Duck Typing**。下面是来自西方的一句格言。

If it walks like a duck and quacks like a duck, it must be a duck（走起路来像鸭子，叫起来也像鸭子，那么它就是鸭子）。

从这里可以导出这样的规则：如果行为像鸭子，那么不管它是什么，就把它看做鸭子。根本不考虑一个对象属于什么类，只关心它有什么样的行为（它有哪些方法），这就是 **Duck Typing**。提出 **Duck Typing** 这个概念的是大名鼎鼎的专家程序员 **Dave Thomas**。

我们来看一个 **Duck Typing** 的具体例子。假设有一个例程 `logs_puts()`，向文件输出日志消息。假定这个方法有两个参数（输出对象和要输出的消息）。如果是静态类型编程语言，比如 C++，程序会是像下面这样的。

```
void log_puts(ostream out, char* msg);
```

`log_puts()` 例程向输出 `out` 里输出时刻和消息。调用这个例程如下所示。

```
log_puts(cout, "message");
```

`cout`（C++的标准输出设备）上会输出如下日志。

```
2005-06-16 16:23:53 message
```

现在，如果我们想把 `log` 输出到字符串而不是文件的话，那该怎么办呢？

因为指定输出对象的 `out` 参数的类型已经定义成 `ostream`，无法简单地变更，结果我们要么把 `logs_puts()` 这个例程全部复制一遍，另外新增一个以字符串为输出对象的例程，要么先输出到临时文件，然后再把它读到字符串里，没有别的办法。

那么，如果使用 **Duck Typing** 的话，会变成怎样灵活的代码呢？如下所示。

```
log_puts (out, msg)
```

因为是动态类型，所以程序中不用指定参数的类型，下面的调用会和 C++ 一样向 `STDOUT`（Ruby 的标准输出设备）输出同样的日志。

```
log_puts (STDOUT, "message")
```

好了，和刚才一样，现在我们想把信息输出到字符串，有了 **Duck Typing** 就简单多了。任何对象，如果它拥有和输出对象（标准输出设

备) 相同的方法, 那么就可以用它作为输出对象。

Ruby 字符串有和文件一样的输入输出类 **StringIO**。图 2-34 演示了用 **StringIO** 来实现输入输出。

```
# 使用StringIO 类库
require 'stringio'

# 生成StringIO 对象
out = StringIO()

# 和文件一样输出
log_puts(out, "message")

# 表示字符串结果
puts out.string
```

图 2-34 Duck Typing 的例子, 使用 Ruby 的 **StringIO** 类

StringIO 类和 **STDOUT** 的类 (**IO**) 没有继承关系。但是, **StringIO** 类中有 **IO** 类的所有方法。所以, 几乎在所有的情况下, **StringIO** 可以像 **IO** 一样地来使用。

如果用静态语言实现相同功能, 需要首先定义一个具有 **log** 输出功能的类 (在 **Java** 中是接口), 然后将它定义为 **log_puts** 第一个参数的类型。像这个例子, 如果输出对象的类型是编程语言中既有的类型, 那就需要重新定义另外一个对象来表达输出对象。即使是在刚开始编写程序的时候就采用这种机制也很费事, 而如果是在中途才开始引入的话, 程序到处都将需要大规模修改。

使用静态类型语言, 程序员通过类型定义提供了大量的信息, 错误可以尽早检测出来, 程序确保可以执行。其代价是, 如果类型设计的前提发生了变化, 为保证各种类型的一致性, 所有关联的部分都要修改。动态类型语言因为开始就不需要定义数据类型, 所以适应类型变化的能力比较强。

那么, 动态类型语言用 **Duck Typing** 的概念设计时要遵循什么原则呢? 基本原则只有一个, 最低限度是只要掌握下面这个基本原则应该就没有问题了。

2.5.8 避免明确的类型检查

有时需要在程序中检查参数的数据类型。例如图 2-35，如果希望处理对象是字符串，自然就会想在不是 `String` 类对象的时候，抛出异常，报告错误。

```
if not obj.kind_of?(String)
  raise TypeError, "not a string"
end
```

图 2-35 进行明确类型检查的例子，在变量不是 `String` 类的时候，抛出异常

但是如果要用 `Duck Typing` 的概念来实现程序的话，怎么也要忍着点，不要把程序写成这样。如果以类为基准进行数据类型检查的话，就会像静态编程语言一样失去灵活性。无论如何都想检查的时候，也不要检查对象是否属于某个类，而是要检查对象是否有某个方法（参见图 2-36）。

```
if not obj.respond_to?("to_str")
  raise TypeError, "not a string"
end
```

图 2-36 用方法来检查数据类型，只接受有 `to_str` 方法的对象

其实即使不检查方法，如果处理对象不是程序所期待的对象，也肯定会出现找不到方法错误。

2.5.9 克服动态类型的缺点

动态类型的缺点主要有三个，即在执行时才能发现错误、读程序时可用的线索少，以及运行速度慢。

首先，执行时才能发现错误这一点可以用完备的单元测试来解决。如果能严格实行完备的单元测试的话，即使没有编译时的错误检查，程序的可靠性也不会降低。

其次，读程序时可用到的线索少这一点可以通过完整的文档来解决。Java 有 `JavaDoc` 技术，Ruby 也有 `RDoc` 技术，可以在源代码中同时写文档，减轻维护文档的负担。

最后，运行速度慢这一点，随着计算机性能的提高已经不再重要，现在的程序开发中，程序的灵活性和生产力更为重要。

2.5.10 动态编程语言

现在我们对程序开发生产力的要求越来越高。也就是说，要在更短的时间内开发出更多的功能。

开发周期短，就要求我们在开发过程中不断探求最合适的开发方法。这又被称为“射击移动的目标”。像以前那样，一开始把所有情况都考虑到，在确定了需求之后再进行开发的方式已经越来越行不通了。尽快着手开发，快速应对需求变更的开发方式变得越来越重要。

在这种快速开发模式中，`Duck Typing` 所代表的执行时的灵活性就非常有用。Ruby、Python、Perl 和 PHP 等优秀的动态类型编程语言，因为它们在执行时所具有的灵活性而越来越受到人们的关注。

2.6 元编程

“元”一词来源于希腊语中表示“.....之间，.....之后，超越.....”的前缀“`meta`”，有“超越”和“高阶”等意思。在 Ruby 和其他一些面向对象编程语言中，类的类称为元类，支撑别的对象的类对象称为元对象。

元编程是对程序进行编程的意思。也许会让人感觉没什么用。初看起来，的确有些让人摸不着头脑，下面就来一窥元编程的威力吧。

2.6.1 元编程

首先我们看一个 Ruby 元编程的例子。这是一个动态生成方法的示例。

在 Ruby 类中内嵌的 `attr_accessor` 方法模块可以动态生成访问实例变量的方法（参见图 2-37）。在图 2-37 简短的程序中，给 `Person`

类自动生成了 **name** 方法和 **age** 方法，也可以用它们来赋值。

```
class Person
  attr_accessor :name, :age
end
```

图 2-37 元编程的例子。Ruby 使用 `attr_accessor` 生成访问实例变量的方法（这里是 `name` 和 `age`）

重要的是，`attr_accessor` 并不是 Ruby 中的一个语句，而是 **Module** 类提供的一个方法，也就是说，如果你愿意的话，也可以自己来定义具有类似功能的方法。

`attr_accessor` 内部进行如下处理。

1. 对所有的参数作以下的处理。
2. 生成与参数名同名的方法。用该方法可以访问“@参数名”这个实例变量的值。
3. 生成参数名后加“=”的方法。该方法有一个参数，它把参数的值赋给“@参数名”这个实例变量。

以上步骤看起来很简单，但是在 C 或者 C++ 语言中是很难实现的。因为在 C++ 等程序执行时，是不能动态地给类增加方法的。Ruby 可以像图 2-37 这样简单地实现这一功能。实际上 `attr_accessor` 是用 C 语言写成的，如果用 Ruby 自己来写这个方法的话，会像图 2-38 的程序那样。

```
class Module

  def attr_accessor(*syms)
    syms.each do |sym|
      class_eval %{
        def #{sym}
          @#{sym}
        end
        def #{sym}=(val)
          @#{sym}=val
        end
      }
    end
  end
end
```

```
    }
  end
end
end
```

图 2-38 用 Ruby 自己来实现 attr_accessor 的例子

`class_eval` 方法接受字符串参数，在类的上下文中对字符串进行处理。在图 2-38 中，从 `%{` 到 `}` 之间的字符串作为参数传递给 `class_eval` 方法。字符串中 `#{` 和 `}` 之间是可以替换的标识符，会被展开成 `sym` 参数所代表的方法名，每个循环定义两个方法。因此，下面的调用会在被调用的对象类中生成两个方法：一个方法是 `name`，用来访问实例变量 `@name` 的值；一个方法是 `name=`，用来给实例变量 `@name` 赋值。

```
attr_accessor :name
```

不支持元编程的编程语言实现这样的功能是很麻烦的。要么需要扩展语言的语法，要么用宏定义等预处理的方法来实现。无论怎样，在普通语言中这都会很麻烦。

2.6.2 反射

下面说明一下元编程的反射（`reflection`）功能。`reflection` 这个英语单词是反射、反省的意思。在编程语言中它是指在程序执行时取出程序的信息或者改变程序信息。

表 2-3 列出了 Ruby 的反射功能，包括取得变量或方法，取得或变更值等，很丰富。比如实现 `Mix-in` 的 `include` 并不是 Ruby 的语法，而是通过方法来实现的。所以说 Ruby 彻底实现了对程序的动态操作。

表2-3 Ruby的反射功能

功 能	方 法 名
列出类/module 的方法	Module#instance_methods
列出对象的方法	Object#methods

列出对象的实例变量	Object#instance_variables
列出全局变量	global_variables
列出局部变量	local_variables
列出类/module 的常量	Module#constants
获取常量值	Module#const_get
设置常量值	Module#const_set
删除常量	Module#remove_const
列出类变量	Module#class_variables
获取类变量值	Module#class_variable_get
设置类变量值	Module#class_variable_set
删除类变量	Module#remove_class_variables
定义类方法	Module#define_method
删除类方法	Module#remove_method
解除类方法定义	Module#undef_method
给类方法赋予别名	Module#alias_method
包含模块	Module#include
获取父类	Class#superclass
获取包含的类	Module#included_modules
获取类的继承关系	Module#ancestors
给继承设置钩子处理	Class#inherited
给包含设置钩子处理	Module#included
给方法定义设置钩子处理	Module#method_added
给特别方法定义设置钩子处理	Module#singleton_method_added
给未定义的方法设置钩子处理	method_missing
解释字符串	eval
在对象的上下文中解释字符串	Object#instance_eval
在类的上下文中解释字符串	Module#class_eval
检查是否有定义	defined?

现在我们来看一下用这些功能到底都能实现些什么。

2.6.3 元编程的例子

首先看一下反射的例子。

有时我们需要把对一个对象的调用委派给另外一个对象。Ruby 用 **Delegator** 这个库实现了委托功能。**Delegator** 对象中包含有方法委托的对象，把方法调用委派给委托的对象。它实现了设计模式中 Proxy 模式的基础部分。要使用 **Delegator** 功能，可以用 **SimpleDelegator** 这个类。

```
require 'delegator'
d = SimpleDelegator.new(a)
```

只用这两句就可以实现，调用对象 **d** 的方法时可以转变为调用对象 **a** 的方法。仅仅是委派的话也没有什么让人高兴的，实际上我们可以给这个对象增加特异方法¹ 来改变它的部分行为，这就大大扩展了它的应用范围。

1 所谓特异方法，是指类中没有定义而只存在于实例（对象）中的方法。Ruby 以外的其他语言也会用到。

这种处理在 Java 中如何实现呢？在静态类型编程语言 Java 中，因为需要匹配类型，所以要另外生成一个 **Delegator** 类，专门对应 **a** 的类型来传送每个方法调用。**a** 的方法如果很多的话，这将是很烦琐的工作。恐怕需要用专门的工具来自动生成才行。

而 Ruby 通过动态类型的反射功能第一个实现了 **Delegator**。

2.6.4 使用反射功能

让我们来看看 Ruby 是怎样用反射功能来实现 **Delegator** 这个类的。图 2-39 是 **SimpleDelegator** 类的部分代码。为了容易理解，例子中程序被大幅度简化了。

```
class SimpleDelegator
  #(a)方法的初始化
  Preserved = ["__id__", "object_
  id", "__send__", "respond_to?"]
```

```

instance_methods.each do |m|
  next if preserved.include?(m)
  undef_method m
end

#(b)对象初始化
def initialize(obj)
  @_sd_obj = obj
end

#(c)method_missing
def method_missing(m, *args)
  unless @_sd_obj.respond_to?(m)
    super(m, *args)
  end
  @_sd_obj.__send__(m, *args)
end

#(d)方法确认
def respond_to?(m)
  return true if super
  return @_sd_obj.respond_to?(m)
end
end

```

图 2-39 SimpleDelegator 的代码

图 2-39 的程序分为 4 个部分。首先说明最重要的。图 2-39c 是 **Delegator** 的核心部分。**Ruby** 在调用方法时，如果对象不知道这个方法，就会首先调用 **method_missing** 这个方法。

method_missing 的第一个参数 是被调用的方法的名字，剩下的是传给方法的参数。**method_missing** 的默认实现是进行异常处理的，但通过重载，也可以处理未知的方法。接着说明下面的两个处理。

1. 被委派的对象如果不知道这个方法（**respond_to?**），默认的实现会被调用（**super**），发生错误。
2. 知道的话，用 **__send__** 来调用委派对象的方法。

__send__ 是调用委派对象方法的方法。这个方法的别名是 **send**，由于 **send** 容易重名，所以用了 **__send__**。

`SimpleDelegator` 剩下的部分比较容易实现。就像刚才说明的，`SimpleDelegator` 是通过 `method_missing` 来委派方法的。但是 Ruby 的 `Object` 类有很多方法，是个很大的类。实际上 `Object` 类有 40 多个方法。`SimpleDelegator` 是 `Object` 的子类，是知道这 40 多个方法的，也就不能委派。为解决这一问题，(a) 中用 `instance_methods` 获取方法的列表，除了几个必要的方法（`__id__`、`object_id`、`__send__`、`respond_to?`）以外，取消了其他方法的定义。

(b) 用于设定 `SimpleDelegator` 的委派对象。(d) 是为了让 `respond_to?` 可以正常执行，首先用 `super` 检查自己的方法，然后检查委派对象的方法。

2.6.5 分布式Ruby的实现

`Delegator` 将被调用的方法直接委派到其他对象，这一功能在很多领域都有应用。作为一个例子，我们介绍一下 `dRuby` (`Distributed Ruby`，分布式 Ruby)。

`dRuby` 是通过网络来调用方法的库。`dRuby` 可以生成服务器上存在的远程对象 (`Proxy`)，`Proxy` 的方法调用可以通过网络执行。

调用的方法在服务器上的远程对象中执行，执行结果可以通过网络返回。这和 Java 的 `RMI` (`Remote Method Invocation`) 功能比较相似。但是，利用 Ruby 的元编程功能，不用明确定义接口，也可以通过网络调用任意对象的方法。

C++和 Java 的远程调用是用 `IDL` (`Interface Definition Language`) 等语言来定义接口的，自动生成的存根 (`stub`) 必须编译和连接。和这些相比，Ruby 的元编程更简单。

`dRuby` 的最初版本只有 200 多行程序，这也体现了元编程的力量。但是现在 `dRuby` 作为 Ruby 的标准库，已经有 2000 多行的规模了。

2.6.6 数据库的应用

在数据库领域，元编程也很有用。

Web 应用程序框架 Ruby on Rails（也称为 Rails 或 RoR）² 中也应用了元编程。具体地说，在与数据库关联的类库（ActiveRecord）中，利用元编程简单地把数据记录定义为对象。

2 在 Ruby on Rails 中，仅仅从数据库定义就可以自动生成相关的程序和配置文件，非常便利。详情请参阅 Rubyist Magazine（<http://jp.rubyist.net/magazine/?0004-RubyOnRails>）的介绍资料。

图 2-40 是 ActiveRecord 定义数据库的例子。然后，数据库中定义了表。图 2-41 中演示了对应 users 表的 User 类。

```
class User < ActiveRecord::Base
  has_one :profile
  has_many :item
end

class Profile < ActiveRecord::Base
  belongs_to :user
end

class Item < ActiveRecord::Base
  belongs_to :user
end
```

图 2-40 用 ActiveRecord 定义的记录

```
CREATE TABLE `users` (
  `id` int(11) NOT NULL auto_increment,
  `login` varchar(80) default NULL,
  `password` varchar(40) default NULL,
  PRIMARY KEY(`id`)
) TYPE=MyISAM;
```

图 2-41 图 2-40 用到的 users 表

从图 2-40 中几行代码可以看出，ActiveRecord 进行如下的处理。

1. 类 **User** 是和以类名的复数形式为名字的表 (**users**) 关联在一起的。
2. 定义了从表的命名空间 (**schema**) 访问记录的方法。
3. 用 **has_one** 、 **belongs_to** 等关联定义，提供了访问关联对象的方法。

之所以能够实现这些处理，都是由于元编程功能让我们可以获取类名，在执行时增加方法。元编程的功能使得 **Rails** 被称赞为生产效率高的 **Web** 应用程序框架。

当然 **Rails** 不是万能的，也不能说比别的应用程序框架都好。但是，**Rails** 最大程度地灵活运用了 **Ruby** 语言的优点，从而确实提高了生产效率。

用 **Rails**，一眨眼的功夫就可以生成一个 **Web** 应用程序，给人印象颇深。

2.6.7 输出XML

最后介绍一下输出 **XML** 文件的类库，由 **Jim Weirich** 开发的 **XmlMarkup**。图 2-42 是一个输出 **XML**（参见图 2-43）的简单程序。

```
require 'builder/xmlmarkup'

xm = Builder::XmlMarkup.new(:indent => 2)
puts xm.html {
  xm.head {
    xm.title("History")
  }
  xm.body {
    xm.h1("Header")
    xm.p {
      xm.text!("paragraph with ")
      xm.a("a Link", "href"=>"http://onestepback.
                                     org")
    }
  }
}
```

图 2-42 XmlMarkup 输出的例子

```
<html>
  <head>
    <title>History</title>
  </head>
  <body>
    <h1>Header</h1>
    <p>
      paragraph with <a
href="http://onestepback.org">a
Link</a>
    </p>
  </body>
</html>
```

图 2-43 图 2-42 的输出内容

Builder::XmlMarkup 和 **Delegator** 同样用到了 **method_missing** 的技巧，通过调用方法从而输出了有标签的 XML。

没有标签的文本必须用 **text!** 命令输出。手工写 XML 是很麻烦的，利用 Ruby 块功能则能很方便地处理。

2.6.8 元编程和小编程语言

到目前为止我们介绍了元编程功能，如果是你，会怎么来利用它呢？

Glenn Vanderburg³ 把它灵活运用到了 DSL（领域特定的语言）领域。DSL 是针对特定领域强化了功能的小规模编程语言。DSL 是很早就有的想法，最近，因为通过 DSL 用户可以强化应用程序的功能或者定制一些功能，所以 DSL 再次得到了关注。DSL 主要需要列于表 2-4 的一些功能。

³ 请参考 Glenn Vanderburg 的 *Metaprogramming Ruby Domain-Specific Language for Programmers* (<http://www.vanderburg.org/Speaking/Stuff/oscon05.pdf>)。

表2-4 小语言需要必备的功能

必要的功能	Ruby的功能
类型 (Type)	○
字面量 (Literal)	○
表达式 (Expression)	○
运算符 (Operator)	○
语句 (Statement)	○
控制结构 (Control Structure)	○
声明 (Declaration)	○ (与实现相关)
上下文相关 (Context Dependence)	○ (与实现相关)
单位 (Unit)	○ (与实现相关)
词汇 (Large Vocabulary)	○ (与实现相关)
层次数据 (Hierarchy)	○ (与实现相关)

Ruby 本来就具备从类型到控制结构这些功能。许多 DSL 小语言往往缺乏其中一些功能，Ruby 反而更好使。还有前面学过的 Ruby 可以利用块自己定义实现控制结构的方法，这也是它的优点之一。

剩下的从声明到层次数据等其他的功能，Ruby 也都可以利用自身的功能来实现。元编程对实现这些功能起到了很大的作用。

2.6.9 声明的实现

我们介绍一下表 2-4 中声明的实现方法。

之前我们介绍了使用 `attr_accessor` 进行元编程的例子。在 Ruby 中 `attr_accessor` 只是一个方法，但是我们也可以把它看做声明。另外，`ActiveRecord` 中的 `has_many` 方法，也可以看做声明，这样的例子还有很多。

Ruby 的方法可以读取或改变程序自身的状态，利用普通的方法调用，可以实现其他编程语言中声明所完成的工作。

从外部来看，Ruby 的方法调用可以省略括号，还可以使用 `foo` 这样的符号来表示名字，这些都使 Ruby 程序看起来像在使用声明一样。

2.6.10 上下文相关的实现

下面讲一下上下文相关。上下文相关是指有些定义只是在一定范围内有效。我们看一下图 2-44 的例子。

```
add_user {  
  name "Charles"  
  password "hello123"  
  privilege normal  
}
```

图 2-44 上下文相关的程序示例

这个例子中，`name`、`password` 等方法只是在 `add_user` 块中才有效。也就是说，在 `add_user` 的外部是看不到这些方法的。

在 Ruby 这一层次上，它把块范围内的方法调用对象换成了 `self`。具体说来，像图 2-45 的例子，使用了 `instance_eval` 方法。

```
def add_user(&block)  
  u = User.new  
  # User 定义了 name、password、  
  # privilege 方法  
  u.instance_eval(&block) if block  
end
```

图 2-45 替换图 2-44 程序上下文的处理

`instance_eval` 方法接受块作为参数，把调用对象置换成 `self` 来执行块。结果，对图 2-44 中块范围内的代码而言，默认的调用对象变成了 `User` 类的对象 `u`。所以，在不指定调用对象而调用方法（如 `name` 等）时，就会调用 `User` 类的方法。

2.6.11 单位的实现

在一般的编程语言中，数值是用标量值来定义的，但这只是数本身。数值的单位需程序员来管理。

然而，DSL 想要处理的不单是数，大都想要处理量。因此我们扩展了一些面向 DSL 的库，以便能处理单位。

例如在 Ruby on Rails 中，**Numeric** 类和 **Timer** 类增加了处理时间单位（基本单位是秒）的方法。比如下面的时间

```
3.years + 13.days + 2.hours
```

表示“3 年 13 天又 2 小时”，以秒为单位就是整数 95 803 200。另外像下面这样：

```
4.months.from_now.monday
```

表示“4 个月后的星期一”。

我写这本书的时间是

```
Mon Dec 12 00:00:00 JST 2005
```

这里只是举了一些与时间相关的例子。因为 Ruby 中可以给现有类自由地追加新方法，所以单位处理功能是很容易实现的。

2.6.12 词汇的实现

DSL 是针对特定领域的，所以在这个特定领域都需要什么处理，需要用词汇来表现。针对特定领域，如果用 Ruby 来定义需要的类和方法，那么就可以认为 Ruby 是这个领域的专用语言。这些类和方法可以称为这个领域的词汇。

借用著名的程序员 Dave Thomas 的话，“所有应用程序的开发过程都可以说是设计语言的过程”。从这个观点来看，开发应用程序就是针对应用程序的问题领域定义各种词汇，最后用这些词汇来描述解决问题的方法。

Ruby 的方法调用和块等具有丰富的表现力，用户可以很自然地用它们来定义词汇。另外如果一开始决定不了要用的词汇，那么像 `Delegator`、`XmlMarkup` 一样，可以用 `method_missing` 这个方法来自动态地追加和利用词汇。

2.6.13 层次数据的实现

最后说明一下表 2-4 最后的层次数据。前面的 `XmlMarkup` 就是层次数据，我们可以再看一下图 2-42。程序本身看起来只不过是以块为参数嵌套调用方法，而外观和功能都漂亮地实现了数据的层次化。

2.6.14 适合DSL的语言，不适合DSL的语言

总的来看，Ruby 是非常适合 DSL 的语言。

首先，调用方法的时候可以省略括号，这些表现的多样性使得程序看起来像是在使用声明一样。DSL 的必要功能很多都用数据构造、设定等声明来定义，所以 Ruby 对声明的支持是很重要的。

其次，元编程功能可以获取并更新程序的信息，所以不必使用预处理和宏就可以实现 DSL 必要的功能。像这样不用对语言进行扩展就可以实现 DSL 的方法也称为“语言内 DSL”。

适合语言内 DSL 的编程语言不只是 Ruby。对 Ruby 影响很大的 Lisp 和 Smalltalk 也和 Ruby 一样适合 DSL。特别是 Lisp，原则上固定语法只有 S 式这种数据结构表现形式，几乎是构成任何语言内语言。

Ruby 可以用 `eval` 处理字符串来生成程序，Lisp 则可以用宏处理列表来实现对程序的处理。所以有人把 Lisp 称为 `Programmable Programming Language`（可编程的编程语言）。

Smalltalk 虽然不像 Lisp 那样极端，但是它的动态性并不比 Ruby 差，另外还具有元编程的功能。Smalltalk 的控制结构本来就是用块来实现的，所以语法扩展也是自然天成的。

反之，有的编程语言，如果不用别的方法就不容易实现 DSL。比如 C++、Java 和 C# 等语言就不能像 Ruby、Lisp 和 Smalltalk 那样实现 DSL。

但这并不是说这些语言中不能使用 DSL 方法，比如说自动生成程序代码。首先定义好 DSL 用的小语言，然后编译成 C++、Java 和 C# 等目标语言。编译器经常用到 Ruby 这种具有优秀文本处理功能的语言。*Code Generation in Action*⁴ 一书讲解了这个主题。

⁴ *Code Generation in Action*, Jack Herrington 著, ISBN 1-930-11097-9。主要讲解 Java 程序的代码生成。代码生成程序都是用 Ruby 写的，所以也可以说它是并未明说的 Ruby 书。

另外一个实现 DSL 方法的例子是解释器。比如开发应用程序时从设计到实现是很复杂的，可以利用固定的语法和类库函数来读取程序。

具体的方法是用 XML、DOM (Document Object Model) 等 XML 处理类库来解释小语言语法。这是 Java 应用程序的配置文件采用 XML 的原因之一。通过 XML 文件，不用每次编译 Java 程序就可以改变配置，定制程序的行为。

这种用法可以把 XML 称为 Java 界的 DSL，或者是 Java 应用程序的脚本语言。

实用主义

前面提到了，面向对象的概念是从仿真和人工智能的知识表现、数据结构等很多领域独立发展出的一些思想互相融合而成的。所以，对于面向对象的理解每个人都可能不同，也有过意见不统一而频繁发生争论的时期。

但是，在21世纪的今天，这种争论已经不多见了。虽然不是很清楚原因，不过以个人所见，我觉得是因为面向对象已经变成了常识性的设计方法。

新的热门话题当然容易引发争论，但对常识性的东西就不会再有人争论了。目前，大多数编程语言都具有面向对象的功能，最具人气的面向对象编程语言是 **Java**。关于面向对象再发生争论的可能性应该没有了吧。

当然，面向对象也分为几个流派。**Smalltalk** 派是发送消息，**C++** 派是数据结构化，还有其他各种折中方案。但是各个流派之间的所谓对立都不过是纸上谈兵，体验了各种面向对象语言之后就会感觉到，其实他们并没有什么真正的对立。就像中国的一句俗语说的，“无论白猫黑猫，抓到老鼠就是好猫”，面向对象编程的实用性已经确定了。

就像 20 世纪 70 年代曾红极一时的结构化编程已经变为常识不再成为话题一样，我以前也预想过面向对象编程早晚也会一样。现在这一预想已经成为了现实。

第 3 章 程序块

3.1 程序块的威力

让我们来说明一下 Ruby 的特色功能之一——程序块吧。Ruby 的程序块是指在方法调用时可以追加的代码块。

比如，在对数组各个元素作循环处理时，请看以下程序。

```
ary.each {|x| puts x}
```

在这个例子中，`{|x| puts x}` 就是程序块。这一行程序就表示把数组中各元素赋值给变量 `x`，调用 `puts` 函数。

程序块功能并非 Ruby 首创。Ruby 只是把其他语言中已经存在的功能，在语法上重新配置了一下。作为程序块功能构想的一部分，下面

我们将把其他语言中的这些功能也一起说明一下。

3.1.1 把函数作为参数的高阶函数

Ruby 程序块功能的原理和高阶函数是一样的。高阶函数是指以函数作为参数的函数。

为了实现高阶函数，编程语言中必须把函数和方法作为数据来处理。比如说，**FORTRAN** 等编程语言就无法实现高阶函数的功能，而 **C** 语言就可以。**C** 语言是通过函数指针来把函数作为一种对象¹ 来处理的。

¹ 在 **C** 语言中，对象是指可以处理的数据。这种数据一般被称为第一类数据（first class data）。

C 语言的库函数中也用到了高阶函数。得到广泛应用的 **qsort(3)**² 函数就是一个高阶函数。**qsort** 函数的参数如图 3-1 所示。其中第 4 个参数是 **compar** 函数的指针。

² **qsort(3)** 是 **UNIX** 使用手册中第 3 部分的 **qsort** 的意思。第 3 部分是库函数。另外，第 1 部分是命令，第 2 部分是系统函数调用。

```
#include <stdlib.h>

void qsort(void *base, size_t nmemb, size_t size, int (*compar)
(const void*, const void*));
```

图 3-1 **qsort(3)** 的 API 的内容，第 4 个参数是 **compar** 函数的指针

在 **qsort** 函数的参数中，**base** 是数组的内存地址，**nmemb** 是数组元素的个数，**size** 是各个元素的大小。**qsort** 的实现用到了快速排序的算法。为了排序，各个元素需要比较大小，**compar** 是比较大小的函数。**compar** 函数有两个参数分别传入要比较的数据，第一个参数值大时返回正值，两个参数值相等时返回 0，第一个参数值小时返回负值。

让我们看一下 `qsort` 的具体使用方法。图 3-2 是使用 `qsort` 的一个程序示例。命令行的参数作为要排序的字符串传递给了 `qsort` 函数³。程序的执行结果如图 3-3 所示。

3 为了使程序示例简单些，`argv[0]` 的程序名也一起参加排序。

```
#include <stdio.h>

int compare(void *a, void *b)
{
    return strcmp(*(char**)a, *(char**)b);
}

int main(int argc, char ** argv)
{
    int i;

    qsort(argv, argc, sizeof(char*), compare);
    for (i=0; i<argc; i++){
        printf("%d: %s\n", i, argv[i]);
    }
    return 0;
}
```

图 3-2 `qsort` 的使用示例

```
$ a.out 3 4 2 5 6 7
0: 2
1: 3
2: 4
3: 5
4: 6
5: 7
6: a.out
```

图 3-3 图 3-2 的执行结果

因为用 `qsort` 来给字符串排序，所以比较函数的参数用到了字符串 (`char*`) 的指针。这种指向指针的指针是有一些难度的。

3.1.2 C语言高阶函数的局限

下面，我们再看一个 C 语言的例子。Ruby 程序有实现哈希表的类库⁴。

4 哈希是使用指定的随机数作为输入数据生成值的方法。

这个类库提供了按顺序操作哈希表各个元素的函数 `st_foreach()`（见图 3-4）。`st_foreach` 为每个哈希元素传递键、值和 `foreach` 的第 3 个参数，一共 3 个参数。第 3 个参数 `arg` 会被再传送给 `func` 这个函数。为什么需要 `arg` 这个参数呢？

```
int st_foreach(st_table *table, int (func*)(st_data_t, st_data_t,
st_data_t),
st_data_t arg);
```

图 3-4 `st_foreach()` 函数的定义，按顺序处理哈希表的各个元素

这是因为在 C 语言中，实现函数间的信息传递只有两种方法：要么明确地传递参数，要么使用全局变量，没有其他方法。

但是如果使用全局变量来传递信息，就搞不清楚什么时候、谁在引用或者更新这一变量，也不能进行递归调用。除了全局变量之外，没有别的办法在函数间共享信息，这是 C 这种语言的局限。

3.1.3 可以保存外部环境的闭包

C 语言中的这种局限在 Java 等语言中也是存在的。而在 Ruby 中，增加了引用函数外部变量的功能。

图 3-5 显示的是从列表中提取出指定长⁵数据的 Ruby 程序⁶。`select` 是把块执行结果为真的数据集中到数组中的方法。所以，`high_paid` 方法可以返回工资是 150 以上的数组。

5 应为指定大小。——译者注

6 图 3-5 的程序引用了 Martin Fowler 的文章 *Closure* 并且做了改编

（<http://martinfowler.com/bliki/Closure.html>，日语网页 <http://capsctrl.que.jp/kdmsnr/wiki/bliki/Closure>）。

```
def high_paid(emps)

  threshold = 150
  return emps.select {|e| e.salary > threshold}

end
```

图 3-5 从列表中提取出指定长数据的 Ruby 程序，程序中用到了块

和上文关于 C 语言指针的例子相比较就不难发现，Ruby 具有以下两个易于使用的优点。

- 可以在使用的时候定义；
- 可以引用外部的局部变量。

当然，Java 的匿名类和 C 语言的函数指针也能实现同样的功能，但是不会像图 3-5 中的程序那样简练。

在块中可以引用外部局部变量的方法，这说明块不只是简单的程序代码，而且把外部“环境”也包括了进来，像这样的块称为闭包。通常的局部变量在方法执行结束时就不存在了，但是如果被包括进了闭包，那么在闭包存在期间，局部变量也会一直存在。

3.1.4 块的两两种使用方法

让我们来详细看一下 Ruby 的块。Ruby 的块是可以追加给调用方法的代码块，块自身并不是对象（对象化后的块是闭包）。参数传递的方法和普通函数不同。仔细看一下下面这一行程序。

```
ary.each {|x| puts x}
```

我们可以得出以下几点：

- 调用了 `ary` 对象的 `each` 方法；
- 没有普通参数；

- 附加了块。

在被调用的方法中有两种方式来使用传递过来的块。一种是用“块参数”的方式明确声明接受块作为参数，另一种是使用 **yield** 这个 Ruby 的保留词。图 3-6 演示了块参数的使用方法，以及用块参数定义数组循环处理的 **each** 方法。

&block 是块参数。块通过闭包的形式被传递给了 **each** 方法的块参数。如果调用时参数中没有块，那么 **nil** 会被作为参数传入。在图 3-6 中，块参数的调用使用了闭包的 **call** 方法。

```
def each(&block)
  i=0
  while i < self.size
    block.call(self[i])
    i+=1
  end
end
```

图 3-6 块作为参数的使用方法 1，定义了块并且调用了 **block.call**

在图 3-7 中用 **yield** 来作块的处理。从表面来看，和用块作为参数的方法有两点不同：

```
def each()
  i=0
  while i < self.size
    yield self[i]
    i+=1
  end
end
```

图 3-7 块作为参数的使用方法 2，用 **yield** 来处理

- 表面上没有定义块参数；
- 用 **yield** 代替了 **block.call**。

使用 **yield** 时，块的信息只是保存在内部堆栈里，并没有用到闭包，所以这种方法的执行速度比参数方法稍微快一些。另外，没有传递块时的错误提示信息也不一样（参见图 3-8）⁷。

7 使用块参数时的错误提示信息不是很好理解。作为编程语言的设计者，我想改进但是还没有好的想法，现公开征集改进方法。

- 块参数版

```
undefined method `call' for nil:NilClass (NoMethodError)
```

- **yield** 版

```
no block given (LocalJumpError)
```

图 3-8 找不到块时的错误提示信息不同

通过对两种方法的比较，我们可以总结出块作为参数具有以下三个优点：

- 明确表示了块处理；
- 块和对象一样被统一处理；
- 检查参数是否为 **nil** 就可以判断出是否传递了块参数。

另外，**yield** 具有下面两个优点：

- 没有用到闭包，执行速度稍快；
- 错误提示信息比较容易理解；

还有，在 **yield** 版本中判断是否传递了块可以用下面的语句。

```
defined? yield
```

3.1.5 最终来看，块到底是什么

Ruby 的块具有以下 3 个特点：

- 代码块可以作为参数传递给方法；
- 在被调用的方法中可以执行传递过来的代码块，执行后程序的控制权返还给方法；
- `n` 块中最后执行的算式的值是块的值，这个值可以返回给方法。

块也可以被看做只是高阶函数的一种特殊形式的语法。虽然只是稍作改进，但 Ruby 中块的各种灵活运用的方法还是让人赞叹不已。

3.1.6 块在循环处理中的应用

最典型的用法是，在逐个处理集合对象的元素的方法中使用块。下面这一行程序相信你已经很熟悉了。

```
ary.each {|x| puts x}
```

Ruby 中几乎所有的容器类都有 `each` 这个方法。使用这个方法可以循环处理容器类中的所有元素。

也可以用 `for` 语句来实现 `each` 方法。

```
for x in ary
  puts x
end
```

这个例子中是在循环内部调用 `puts` 方法，和 `ary.each...` 的动作几乎是一样的。

本来，Ruby 就是为了要实现循环功能才导入了块的。所以，在以前文档中把具有块的方法称为迭代器（`iterator`）。`iterate` 就是循环、迭代的意思。但是，如今块的应用范围比当初所能想到的要广泛得多，和

循环没有关系的处理中也大量地用到了块。所以现在仍把块称为迭代器就很不恰当了。

3.1.7 内部迭代器和外部迭代器

像 Ruby 块这样，把对各个元素的处理逻辑传送给容器类的方法，然后在方法中对容器类中每个元素调用指定的处理逻辑，这种迭代方式称为内部迭代器方式。

与之相对应，C++和 Java 中所谓的迭代器，是用别的类对象来循环处理容器中的元素（参见图 3-9），这种循环处理的方式称为外部迭代器方式。在外部迭代器方式中，把顺序取出容器中元素的对象称为迭代器，也称为游标。

```
for (Iterator i = ary_names.iterator(); i.hasNext();) {  
    Object obj = i.next();  
    ...;  
}
```

图 3-9 Java 的外部迭代器示例

内部迭代器不用额外生成类，使用 and 实现都很简单。但是，对于不支持闭包的编程语言，想要拥有循环外部的信息就要费些工夫，像在 C 语言中使用循环就不太方便。所以，没有闭包功能的 C++和 Java 就采用了外部迭代器方式。

另外，在外部迭代器方式中容器和迭代器关系密切，实现时比较困难，使用时的编程量也比较大，而内部迭代器只用一行程序就可以实现循环了。但是外部迭代器也有它自身的优点。在从多个容器中逐个取出数据进行并行处理的时候，外部迭代器可以简单地处理，而内部迭代器就实现不了。

由此可以看出，外部迭代器和内部迭代器各有所长。但是，如果编程语言支持闭包功能的话，还是用内部迭代器比较方便⁸。

⁸ 从设计模式来看，内部迭代器是访问者模式，外部迭代器是迭代器模式。

3.1.8 在排序和比较大小中的应用

程序块可以像 C 语言的 `qsort` 那样，作为每个元素的判定条件来使用。例如在 Ruby 中，可以采用如下的方式在排序方法中使用块。

```
ary.sort{|a,b| a<=>b}
```

如果和 C 语言中 `qsort` 的用法相比较的话，我们就可以看到 Ruby 中块的用法是多么简单。最初，如果在 `sort` 方法中没有指定块，那么元素的比较默认也是用 `<=>` 算式的，所以这里的指定和默认动作是一样的，没有太大意义。

现在我们把各个元素变换成整数来排序。

```
ary.sort{|a,b|  
  a.to_i <=> b.to_i  
}
```

这也很简单。在没有指定块的时候，是按照字典顺序来比较的。现在像这样指定了块，那么就按照数值顺序来进行比较。按字典顺序比较时，10 是排在 1 和 2 之间的，按数值顺序比较时则排在 9 的后面。

仔细想想，如果每次比较都要做整数变换处理的话，是很浪费资源的。排序时要多次进行比较处理，比较的次数随着元素个数的增加而增加。现在的 Ruby 排序中，元素有 4 个的时候需要比较 3 次，100 个元素需要比较 500 次，1000 个元素则需要比较 9500 次。所以当元素特别多的时候，执行块的代码来进行比较处理的代价就不能忽视了。

针对这种情况，可以用 `sort_by` 方法。`sort_by` 用执行块的代码所生成的结果来排序，对每个元素只执行一次块的调用。

```
ary.sort_by{|x| x.to_i}
```

3.1.9 用块保证程序的后处理

Ruby 和别的编程语言一样，有异常处理功能，在发生错误时处理可能会中断。比如，下面的文件处理程序。

```
f = open(path)
...#(a)
f.close
```

如果(a)的部分发生了异常，但文件可能没有被关闭。在这种情况下，就需要用 Java 中与 **finally** 同样功能的 **ensure** 来保证文件被关闭。

```
f = open(path)
begin
  ...
ensure
  f.close
end
```

这样就变得很安全了，但是如果每次都这样写的话就会很烦琐。现在，让我们来看看块是怎样来解决这个问题的吧。

在 Ruby 中，**open** 可以用如下的写法来表示。

```
open(path) {|f|
  ...
}
```

如果给 **open** 方法传递了块，那么在结束时文件就可以自动关闭。

3.1.10 用块实现新的控制结构

如果用块的话，不需要改变文法，就可以实现控制结构的定义。比如在无限循环的 **loop** 或者指定次数的循环 **times** 中，用块来实现控制

结构（参见图 3-10）。

```
loop {  
  #直到中断（break）为止，  
  #否则块无限循环  
}  
  
3.times {  
  #3 次循环  
}
```

图 3-10 times 方法的示例，用块实现控制结构

块还可以用于指定条件。比如容器类的 **reject** 方法把块处理中结果为真的数据删除掉，返回剩余元素的数据（参见图 3-11）。

```
ary = [1,2,3,4]  
#删除偶数  
ary.reject {|x| x%2 == 0}  
# => [1,3]
```

图 3-11 reject 方法的示例，用块来指定条件

在图 3-12 中，用循环来实现同样的功能。**reject** 只需要一行程序，循环却需要 6 行程序。此外，**reject** 程序不仅很短，而且意图很明确。

```
result = []  
for x in ary  
  if x%2 != 0  
    result.push x  
  end  
end
```

图 3-12 用循环来重写图 3-11 中的程序

像这样的方法 Ruby 中还有好几个，其方法名大都以 **-ect** 结尾。表 3-1 列出了从 Smalltalk 继承的方法名。

表3-1 和 reject 类似的方法

方法名	功 能
collect	集成块的结果
select	集成块结果为真的元素
detect	返回块结果为真的最初元素
reject	集成块结果不为真的元素

3.1.11 在回调中使用块

块也可以在回调中使用。图 3-13 是利用 Tk⁹ 的 GUI 程序。

⁹ Tk 是 UNIX 中的 GUI 工具包。

图 3-13 的程序执行时会显示 **hello** 按钮。按下的话，会在标准输出中显示 **hello** 字符串。

```
require 'tk'

#按钮生成
b = TkButton.new(:text => "hello").pack
#按下的动作
b.command{ puts "hello" }
#消息循环开始
Tk.mainloop
```

图 3-13 回调中使用块，利用了 Tk

`command` 方法指定的块，会作为一个闭包被保存在按钮对象里，按下时被执行。

3.1.12 块处理的特别理由

如上所述，Ruby 的块具有以下特点：

- 在普通参数以外，另外被传送；

- 块不是对象（**lambda** 方法可以作闭包对象化）。

其他具有闭包功能的编程语言，比如 **Lisp** 和 **Smalltalk**，它们没有这样的区别，总是把闭包作为对象来处理。从这一点来看，**Ruby** 是作了改进。这到底是为什么呢？

这有两个理由。一个是减少对象的生成数。初期 **Ruby** 生成闭包对象的代价很高，所以尽量避免了闭包对象的生成。即使是真正必要的对象，也尽量延迟到必要的时候才生成，我通过这种方式努力提高程序性能，哪怕是只能带来一点点的改善。

另一个是外观上的理由。如果把块做成和 **Lisp** 或者 **Smalltalk** 闭包一样的话，那么 **each** 的外观就会像下面一样。

```
ary.each ({|x|puts x})
```

这样看上去就不像是一个控制结构。**Ruby** 之所以不用扩张文法就可以很自然地追加控制结构，是因为在调用方法时，对块有着特别的处理。

在 **Ruby** 块的设计中，考虑到了和其他语句的协调。我们用别的具有块或者高阶函数的编程语言来比较一下。比如在 **Smalltalk** 语言中，所有的控制结构都用块来表示。**if** 语句的写法如下所示。

```
(age < 18) ifTrue: [ adult := true ].
```

Smalltalk 中方括号里面的部分是块。这一行程序是用 **ifTrue** 来判断其真假的，只有当条件为真的时候，后面的块才会被执行。

另外，**while** 语句的写法如下所示。

```
[i < 10] whileTrue: [ i := i + 1 ].
```

这个例子里，前面的条件也是一个块。这行程序是在调用第一个块[`i < 10`]的 `whileTrue:` 方法。

后面的块[`i := i + 1`]是这个方法的参数。`whileTrue:` 方法会在第一个块为真时反复执行第二个块。

和 `if` 不同，因为 `while` 语句的条件判断也要反复执行，所以这一个部分就不能是单纯的表达式，而需要把它写成块。不客气地说，`Smalltalk` 把循环处理也都作为方法调用，为了简化语法而区别使用一般的条件判断表达式和块，结果是使用时不是很方便。

我们再来看看在循环中多用高阶函数的 `Lisp` 编程语言。`Ruby` 的 `each` 在 `Lisp` 中是像下面这样的。

```
(foreach ary (lambda (x) (puts x)))
```

如果用 `Ruby` 来写的话，是像下面这样的。

```
ary.each{|x| puts x}
```

先不用说 `Lisp` 中需要很多括号这个特征，为了做成闭包，`lambda` 也是必不可少的，这就让人觉得它只是一个高阶函数，而不是一个控制结构。

`Lisp` 中如果用宏来追加控制结构的话，程序是像下面这样的。

```
(foreach x (puts x))
```

因为用宏置换掉了程序，所以就看不出来内部有没有使用高阶函数。`Lisp` 中高阶函数就是高阶函数，控制结构扩张时就用宏，分工是不一样的。所以，同样是用闭包的语言，从语法的不同到整个编程语言风格的不同，都是很有趣的。

虽然有点“王婆卖瓜，自卖自夸”的意思，但 Ruby 这种不用改变语法就可以追加新控制结构的功能很是漂亮。当然，调用方法时只能使用一个块，是 Ruby 中的一个限制，但实际情况中也几乎没有必要使用多个块。

3.2 用块作循环

Ruby 的块本来就是在循环的抽象化过程中诞生的。现在除了循环以外，在其他一些场合也得到广泛应用，但这并没有改变其实现循环的初衷。这里我们再次说明一下如何用块来实现循环。

3.2.1 块是处理的集合

循环是程序的基本元素。从结构化编程的原理上说，所有的算法都是由顺序、分支和循环的组合来实现的。可以说处理好了循环，也就处理好了程序。

Ruby 和其他编程语言一样，条件成立时的循环也用 **while** 来表现。在这之外，循环还有其他更为深奥的表现形式。

首先，Ruby 中有 **until** 语句。**until** 语句是直到条件成立时为止一直循环，和 **while** 语句正好相反。如果只是 **until** 语句的话，并不足为奇，重要的是 Ruby 还可以处理块，这是它独有的特点。

块是处理的集合。Ruby 中，在方法调用的最后，可以附加上块，比如下面的例子。

```
ary = [1,2,3]
ary.each{|i| puts i}
```

第 2 行大括号中的部分是块。这行程序是用块作为参数来调用数组的 **each** 方法。

each 方法可以对数组的各个元素进行循环处理。夹在“|”中间的变量 **i**，可分别赋值为“1、2、3”中的各个元素来进行块的处理。

块中的 **puts** 是把对象加上换行输出到标准输出设备的函数。在这里，执行代码后，会一行一行地输出 1、2、3。

这里重要的是，**each** 方法中根本不包含“怎样循环”的信息。也就是说，**each** 方法与数组内部的详细实现完全无关。

所以说，无论数组内部结构是否变化，或者是换成数组以外的数据结构，对 **each** 方法的处理都没有影响。因为 **each** 方法表达的是“对各个元素进行循环处理”这一本质部分，所以上述变化对它没有影响。

像这样，我们把和数据内部详细实现无关而只是对各个元素进行循环处理的方法称为循环的抽象化。**Ruby** 的简洁性，不只是体现在程序简短，更重要的是体现在对本质问题的处理，使得程序更为灵活。

用 **Ruby** 实现这种循环，实际上非常简单。只是在块调用的地方，像下面的程序一样用 **yield** 来指定而已。**yield** 的意思是转让。

```
def ary_each(ary)
  i = 0
  while i < ary.size
    yield ary[i]
  end
end
```

也可以用 **do~end** 来定义块。大括号和 **do~end** 基本上是一样的。但是，在块是多行的时候，用 **do~end** 看起来和别的结构的统一性更好一点。刚开发 **Ruby** 的时候，块的定义只有大括号，因为右边的大括号和 **end** 混在一起的时候看起来很别扭，所以增加了 **do~end** 语句。它们两个使用方法的区别如下：

- 块是一行的时候用大括号，是多行的时候用 **do~end**；
- 块作为表达式的一部分，给方法返回值时用大括号，块作为处理语句或程序流程控制的时候用 **do~end**。

另外，大括号和 **do** 的结合优先级不一样。大括号的优先级更高。在省略参数的括号时，这种区别就会显现出来了。比如下面的程序，

```
a b {|x| puts x}
```

因为大括号的优先级高，所以就认为块是和 **b** 结合在一起的。加上括号之后的程序如下所示。

```
a(b {|x| puts x})
```

反之，在下面的程序中，因为 **do** 的优先级低，所以 **b** 是和 **a** 结合在一起的。

```
a b do|x| puts x end
```

那么，加上括号之后的程序如下所示。

```
a(b) do|x| puts x end
```

因此，在省略括号调用方法时，一定要注意块的结合优先顺序。

3.2.2 块应用范围的扩展

现在，块被广泛应用于各种各样的领域中，但最早起源于在循环中的应用。其实，我在最早介绍 Ruby 的《面向对象的脚本语言 Ruby》（1999 年）一书中，把调用块的方法称为迭代器。迭代器就是循环的意思。所以当时块主要是用在循环里的。

Ruby 块起源于 20 世纪 70 年代麻省理工学院（MIT）开发的 CLU 编程语言。CLU 中有迭代器功能，用于循环的抽象化（参见图 3-14）。

```
for i:int in int$from_to(1,string$size(s)) do  
  ...  
end
```

调用迭代器的部分

图 3-14 CLU 中的迭代器

图 3-14 中的 `int$` 是调用整数功能的意思。在这个迭代器中，从 1 开始到 `string$size(s)` 为止，循环做 `do` 和 `end` 之间的处理。

CLU 的迭代器是在 `for` 语句中才能调用的特殊例程。该例程用 `yield` 语句传递循环变量，图 3-14 中的 `i`，用于实现循环处理。循环结束后，程序继续执行 `yield` 之后的语句。

从根本上来看，这和 Ruby 的块是相似的。但是因为在 CLU 中只能用 `for` 语句来调用迭代器，所以它的用法就受到了限制。

在 Ruby 中，不必用特别的结构，在任意的方法中都可以使用块，所以不仅仅是在循环中，块在各种各样的领域中都得到了应用。从语法上的一个微小区别到应用领域中的巨大差别，这之中应该有很多地方值得我们深思吧。

3.2.3 高阶函数和块的本质一样

Ruby 的块本质上和高阶函数是一样的。高阶函数是接受函数作为参数的函数。

编程语言要实现高阶函数，就必须把函数或者方法作为数据来处理。反之，具有这样功能的编程语言就可以利用高阶函数。

例如，C 语言可以把函数作为指针来处理，所以可以实现高阶函数。在前面所举的 `each` 方法的例子中，我们可以把块看成是具有“`puts` 参数”功能的匿名函数，把这个匿名函数作为参数传递给 `each` 方法，然后在 `each` 方法中调用这个匿名函数，这就跟高阶函数是一样的了。

如果把 Ruby 的块看成是高阶函数来调用的话，那么在语法上就有限制，只能有一个函数参数。而这带来的好处是，使用块可以自由地扩展语言的控制结构。

说一下题外话，这里有一个有趣的调查结果。在倾向于使用高阶函数的 OCaml 编程语言的 2239 个库函数中，没有用函数参数的占 87.2%，用一个函数参数的占 12.1%，用两个以上函数参数的只占 0.7%。

所以，在高阶函数中，94%都只有一个函数参数，有两个以上函数参数的是极少数的。Ruby 以更易于使用的形式，把只有一个函数参数的情形在语法上加以特殊处理，导入了块功能，这在设计上是没有问题的，真是太好了。

3.2.4 用 Enumerable 来利用块

虽然块的应用范围不局限于循环，但我们还是重点从循环讲起吧。

把块应用于循环抽象化的，最典型的应该是 Enumerable 这个模块。Enumerable 模块以 each 方法为基础，为定义 each 方法的类提供了多种功能。如果继承（Mix-in）了这个模块，就可以很方便地利用它的各种功能。表 3-2 中列出了 Enumerable 提供的方法。

表3-2 Enumerable 的方法

方 法	说 明
all?	是否全部为真
all?{ x ...}	块是否对所有的元素都为真
any?	是否有元素为真
any?{ x ...}	块是否对有的元素为真
collect { x ...}	块对各个元素处理结果的数组
detect { x ...}	结果为真的第1个元素
each_with_index { x, i ...}	按照下标顺序对各个元素处理
each_cons(n) { x ...}	分别对前后n个元素进行块处理（有重复）（Ruby 1.8.7）
each_slice(n) { x ...}	分割成n个元素后处理（Ruby 1.8.7）
entries	元素的数组
find { x ...}	块为真的第1个元素
find_all { x ...}	块为真的元素数组
grep(pattern)	模式匹配的元素数组
grep(pattern) { x ...}	对模式匹配的元素数组进行块处理
include?(x)	和x相等的元素是否存在
inject { x, y ...}	元素的块处理结果
inject(init) { x, y ...}	元素的块处理结果

<code>map { x ...}</code>	元素的块处理结果的数组
<code>max</code>	最大的元素
<code>max { a, b ...}</code>	用块比较出的最大的元素
<code>max_by { x ...}</code>	用块变换后的最大的元素 (Ruby 1.8.7)
<code>member?(x)</code>	和x相等的元素是否存在
<code>min</code>	最小的元素
<code>min { a, b ...}</code>	用块比较出的最小的元素
<code>min_by { x ...}</code>	用块变换后的最小的元素 (Ruby 1.8.7)
<code>partition { x ...}</code>	用块把真假元素分离
<code>reject { x ...}</code>	块处理结果为假的元素集
<code>select { x ...}</code>	块处理结果为真的元素集
<code>sort</code>	对元素排序
<code>sort { a, b ...}</code>	用块处理来排序
<code>sort_by { x ...}</code>	用块处理变换后的结果来排序
<code>to_a</code>	元素的数组
<code>zip(a, ...)</code>	连接各个集合到数组
<code>zip(a, ...) { arr ...}</code>	连接各个集合后执行块

Enumerable 的意思是可数的。它是对数组等各种集合的元素做循环处理的方法的集成。这些方法大致可以分为以下几类：

- 循环
- 指定条件
- 排序、比较大小

我们来看看 **Enumerable** 的功能和这些功能的使用方法吧。

循环

循环处理的基本方法是 **each**。在下面的程序中，假定 **col** 是个集合对象，首先把各个元素值赋值给变量 **x**，然后执行 **puts x**。

```
col.each {|x| puts x}
```

必须要注意的是，`each` 并没有在 `Enumerable` 中定义。反过来，`Enumerable` 中的所有方法都是在内部调用 `each` 来实现的。

只要是可以用 `each` 方法对各个元素进行循环处理的对象，用 `include Enumerable` 就可以使用表 3-2 中所表示的各种方法了。

循环方法中最简单的是 `each_with_index` 方法。它把元素和下标一起传递给循环处理方法。

```
["a", "b", "c"].each_with_index{|x, i|
  printf "%d: %s\n", i, x
}

# 输出
# 0: a
# 1: b
# 2: c
```

最常用的方法之一是 `collect`。`collect` 是对各个元素执行块处理，返回处理结果的数组。块的执行结果就是块中最后一个表达式的值。比如下面的程序：

```
[1, 2, 3].collect{|x|
  x*2
}

# 结果
# [2, 4, 6]
```

在 Ruby 的 `Enumerable` 中，像 `collect`、`select` 和 `detect` 等方法一样名字以 `-ect` 来结尾的方法有很多。这些方法名是继承自 `Smalltalk` 的方法名。`collect` 的别名 `map` 也是继承自 `Lisp` 编程语言中的方法名。

zip 是把多个集合中的元素同时取出来的方法。具体请看下面的例子。

```
a = [1,2,3]
b = [2,4,6]
a.zip(b)
# 结果
#[[1,2],[2,4],[3,6]]
```

不用块来调用 **zip** 时，它返回一个数组，数组中第 n 个元素是由每个集合的第 n 个元素所构成的数组。另外，如果用块来调用 **zip** 的话，它就把每个集合的第 n 个元素一起传递给循环处理。

```
a = [1,2,3]
b = [2,4,6]
a.zip(b) {|x,y|
  printf "[%s,%s]\n",x,y
}
# 输出
# [1,2]
# [2,4]
# [3,6]
```

grep 方法可以对集合中的元素进行模式匹配。匹配用 `===` 运算符来表示。没有指定块的时候，返回集合中匹配元素（`===` 运算符返回真）的数组。如果指定了块，和 **zip** 一样，就把各个匹配的元素传递给块来处理。

```
a = ["foo","bar","baz"]

a.grep(/^b/)
#结果是以b 开头的元素
#["bar","baz"]

a.grep(/^b/) {|x|
  printf "%s\n",x
}
# 输出
# bar
# baz
```


条件指定

对集合的各个元素进行块处理的是循环型的方法。和它相对的，对各个元素进行块处理，把块处理的结果作为下个动作的判定条件的方法是条件指定型方法。

条件指定型方法中最常用的是 **select** 方法。**select** 方法把块处理结果为真的元素存放在数组中返回。这个方法别名是 **find_all**。

```
a = [1,2,3,4]
a.select{|x| x%2 == 0}
# 结果 (偶数)
# [2,4]
```

和 **select** 相反的方法是 **reject**。**reject** 方法返回块处理结果为假的元素的数组。

```
a = [1,2,3,4]
a.reject{|x| x%2 == 0}
# 结果 (奇数)
# [1,3]
```

如果想找出第一个满足条件的元素，就可以用 **detect** 方法。它的别名是 **find**。

```
a = [1,2,3,4]
a.detect{|x| x%2 == 0}
# 结果 (第1 个偶数)
# 2
```

真假结果都需要的情况也不能说没有。这样一个八面玲珑的方法是 **partition**。**partition** 方法返回真的和假的元素的数组。

```
a = [1,2,3,4]
a.partition{|x| x%2 == 0}
# 结果
# [[2,4],[1,3]]
```

全部元素是否都为真，或者至少有一个元素为真的判断方法是 **all?** 和 **any?**。

```
a = [true,true]
b = [false,true]
a.all? # => true
b.all? # => false
a.any? # => true
b.any? # => true
```

all? 和 **any?** 可以用块处理的结果作为判定条件。

```
c = [1,2,3]
c.all?{|x| x%2 == 0} # => false
c.any?{|x| x%2 == 0} # => true
```

排序与比较大小

有一些方法是用对块操作的结果来比较大小。

其中 **min** 和 **max** 返回集合中最小和最大的元素，如果指定有块的话，就用块来比较两个元素的大小。比较用 **<=>** 运算符，***a* > *b*** 的时候返回正数，***a* = *b*** 时返回零，***a* < *b*** 时返回负数。

```
ary = ["1","11","2","6"]
p ary.max
# => "6"
p ary.max{|a,b| a.to_i <=> b.to_i}
# => "11"
```

`sort` 方法是把集合的元素按照从小到大的顺序排序，和 `min` 一样可以传递给用于比较大小的块。

```
ary = ["1", "11", "2", "6"]
ary.sort{|a,b| a.to_i <=> b.to_i}
# 结果
# ["1", "2", "6", "11"]
```

但是，块处理的调用是要花费时间的，而且在排序的时候，与其说是比较两个元素，更多的时候是对每个元素进行某种处理（取得某个属性等），用处理的结果来排序。另外，不要每次在比较的时候都调用块处理，先一次性把全部的元素都处理好，然后用处理结果来排序常常会更快。Ruby 的 `sort_by` 方法就是用这种方式来排序的。

```
ary = ["1", "11", "2", "6"]
ary.sort_by{|x| x.to_i}
# 结果
# ["1", "2", "6", "11"]
```

从最新版本的 Ruby 1.8.7（也包括 1.9）开始，Ruby 也提供了跟 `min` 和 `max` 方法一样用块来指定比较方法的 `min_by` 和 `max_by` 方法。

3.2.5 Enumerable 的局限

像上面介绍的那样，`Enumerable` 提供了以循环为基础的方法，但是 `Enumerable` 也有它的缺点，主要的两个缺点是：循环都依赖 `each` 方法，而且不能并行执行。`Enumerable` 可以用 `each` 方法简单地实现循环，但是反过来说，这也是它的一个局限。

比如字符串。对字符串实行循环时，可以想到的组合单位有文字、行或字节。根据不同情况需要的单位可能不一样，所以不能说哪种单位是正确的。`String` 类是字符串，所以可能很多人会觉得它的单位是文字。但是，在 Ruby 1.8 中，它是以行为单位来循环的。在 Ruby 1.9 中，为了强调“没有最好的方案”，我们果断决定 `String` 类不再继承 `Enumerable` 模块。对于循环时要用什么单位，可以分别用

`each_char`、`each_line` 或 `each_byte` 等这些不同的方法来定义。

顺便再说明一下，在 Ruby 1.8 中，和大多数人想象的不同，字符串的 `each` 方法是以行为单位来循环的。

以文字为单位来循环时用 `each_char` 方法（从 1.8.7 版开始提供），以字节为单位来循环时用 `each_byte` 方法。但是，`Enumerable` 模块提供的方法就不能用这些单位来处理了。这就是 `Enumerable` 只依存于 `each` 方法的局限性。

循环不能并行处理这一点是 Ruby 块所共有的局限性。Ruby 的方法接受块参数，在方法内部进行初始化、条件判断等循环处理，再回过头调用块的处理，这种循环处理被称为内部迭代器。这种方法定义简单且容易理解，但是如果从多个对象逐个取出元素来处理的话，就没有别的办法，只有把这所有对象的元素变成一个数组时，才能从第 1 个开始逐个处理。

相反，在 Java 或者 C++ 中，对于向量（`vector`）等循环处理对象，是先从中取出指定循环位置的对象“游标”，然后通过这个对象对各个元素进行循环。这种通过外部对象进行循环的机制称为外部迭代器。外部迭代器对原始对象的依赖性很强，所以其机制很复杂，定义很困难。

另外，调用方每次都要作游标的初始化和条件判断（是否还有下一个元素），然后取出下个元素，所以使用方法变得很复杂。但是这样做的好处是，同时从多个对象取出元素进行处理的程序很好写，而这正是内部迭代器难于处理的情形。

我在最初设计 Ruby 时就已经知道有这些问题，但是因为它们并不频繁发生，所以不是很重视。Ruby 1.8.7 版以后提供了 `Enumerator` 类，同时解决了以上两个问题。表 3-3 中是 `Enumerator` 类（正确地说应该是 `Enumerable::Enumerator` 类）的方法一览。

表3-3 `Enumerator::Enumerator` 的方法

方 法	说 明
<code>each</code>	用指定的方法把元素送给块去处理

with_index	调用块时附加上下标
next	返回下一个元素（没有时则发生异常）
rewind	next 的顺序重新从第1个开始

在 Ruby 1.8.7 版以后，如果不传递块的话，**Enumerable** 模块几乎所有的都返回 **Enumerator**。然后 **Enumerator** 对象提供基于该方法的循环，而不用 **each**。所以，在对字符串以字节为单位循环，求最大字节值时，可以用下面的程序。

```
str.each_byte.max
```

方法中即使没有返回 **Enumerator** 的功能，也可以使用 **enum_for** 方法得到 **Enumerator** 对象。例如，想取得基于 **each_byte** 的 **Enumerator**，可以用下面的程序。

```
str.enum_for(:each_byte)
```

调用块处理时，如果有第几个元素是否存在这样的索引信息的话，就会很方便，比如在数组前后的元素也一起参与计算的时候。对于 **each**，以前提供了 **enum_with_index** 方法，但是，如果对 **each** 以外的方法，比如 **map**，也都要分别提供这样的方法的话，就会没有穷尽的，所以就没有为这些方法提供相应的带索引的方法。但今后，像下面这样：

```
ary.map.with_index{|x,i|  
  [x,i]  
}
```

把块处理省略掉，用 **with_index** 这样的方法，就可以给 **Enumerable** 的所有循环方法传递索引了。

最后，至于同时并行处理，其实 **Enumerator** 也可以作为外部迭代器来使用。首先取得 **Enumerator** 对象，然后调用 **.next** 方法就可以按顺序逐个取出块应该返回的要素。提供循环的一方只需像平常一样定义自己的内部迭代器，**Enumerator** 类会自动把它转换成外部迭代器（内部处理相当复杂）。

比如我们看一下下面的这个 **zip** 方法。**Enumerable** 提供的 **zip** 方法，是按顺序取出各个参数的元素，然后把这些元素作为一个数组传递进来。

```
[1,2,3].zip([2,4,6],[3,6,9]){|x| p x}
# 输出
# [1,2,3]
# [2,4,6]
# [3,6,9]
```

如果用 **Ruby** 来定义 **zip** 方法的话，到目前为止，因为没有提供外部迭代器，所以只能在内部把参数变换成数组来对应（参见图 3-15 的上半部分）。把参数作成数组，有以下两个问题：

```
1 把参数变成数组的版本
module Enumerable
  def zip(*args)
    n = 0
    args = args.map{|a| a.to_a}
    self.each do|x|
      yield [x, *args.map{|a| a[n]}]
      n += 1
    end
  end
end

2 使用外部迭代器的版本
module Enumerable
  def zip(*args)
    args = args.map{|a| a.enum_for(:each)} #(a)
    self.each do|x|
      yield [x, *args.map{|a|
        a.next rescue nil          #(b)
      }]
    end
  end
end
```

```
end
```

图 3-15 Ruby 定义的 `zip` 方法

- 大量的数据变换成数组会造成内存浪费；
- 不能处理无限循环的 `Enumerable` 。

`zip` 处理的本质是要并行取出各参数的元素，所以受到了内部迭代器的制约。

但是，在 Ruby 1.8.7 版以后，由于提供了 `Enumerator` 功能，程序可以写成如图 3-15 的下半部分那样。

首先，1 的部分不要变换成数组，而是用 `enum_for` 方法变换成以 `each` 为基础的 `Enumerator` 。如果是 Ruby 1.8.7，这部分可以通过调用没有块的 `each` 方法来实现，但是如果是用户自己定义的类，调用没有块的 `each` 方法，有可能并不返回 `Enumerator` 。所以用 `enum_for` 方法才是安全的。

2 的部分是从 `Enumerator` 里面把元素一个个取出来。Ruby 的 `zip` 方法的规格是，如果后面已经没有元素了，那么要加入一个 `nil`，所以这里用 `rescue` 来捕捉没有元素时的异常（`StopIteration` 异常），传递 `nil` 。用 `Enumerator` 的版本和不用 `Enumerator` 的版本动作上没有什么区别，参数如果是很大的集合，不用 `Enumerator` 的话，就要把参数变换成数组，从而会消耗大量内存。

3.3 精通集合的使用

本来导入块功能的目的是对集合中的多个对象作循环处理。块功能和集合结合起来就能发挥更大的作用。所以在这里，既作为一个复习，又学习一下 Ruby 集合的使用方法。

集合可以看做是多个对象的容器。Ruby 标准库里面提供了一些集合类。比较典型的是 `Array` 和 `Hash` 类。这里我们以这两个类为中心来讲解。

3.3.1 使用Ruby的数组

Array 本意是整整齐齐排列在一起的东西，数组实际上也很像这个样子。比如包括整数 1、2、3 的数组可以表示成图 3-16 的样子。

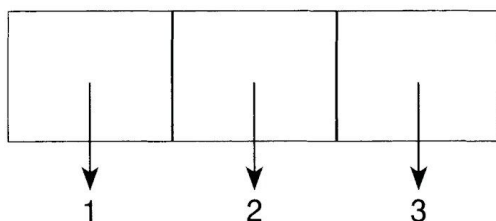


图 3-16 数组的概念

请注意这些整数并不在格子中。刚才说过了，集合是个容器，其实数组中并没有放入对象，而是可以用数组去引用各个对象。

生成数组的方法有多个，最简单的是使用数组表达式。数组表达式是用逗号隔开排列在一起的表达式，并用[]括起来。图 3-16 中数组的定义方法如下所示。

```
[1, 2, 3]
```

在动态语言¹ Ruby 中，集合中可以混合存在各种类型的对象。所以可以定义复杂的数组。

¹ 与 Java 这样的静态语言不同，动态语言总可以调用相同的方法，与继承无关。

```
[1, "two", [3, 4]]
```

上面的例子中，数组的第 1 个元素是 1，第 2 个元素是字符串 **two**，第 3 个元素则是一个数组。

像下面这样引用数组元素。第 1 行程序是初始化，第 2 行是引用。

```
ary = [1, 2, 3]  
ary[0]
```


沿用了 C 语言的习惯，最前面的元素的下标用 0。元素的下标也可以用负数来指定。最后的元素的下标是-1（参见图 3-17）。

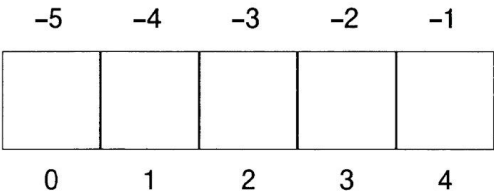


图 3-17 数组的下标

不仅可以取出一个元素，而且可以取出一部分元素。范围的指定方法如图 3-18 所示。请注意..和...的不同，具体请参照图 3-19。

[n,m] 开头元素的位置和元素个数

[n..m] 开头元素和末尾元素的位置（包括末尾的元素）

[n...m] 开头元素和末尾元素的位置（不包括末尾的元素）

图 3-18 取得指定范围的元素的 3 种方法

```
a = [1,2,3]
a[1,2] # => [2,3]  从下标为 1 的元素开始取两个元素
a[1..2] # => [2,3] 取下标为 1 到下标为 2之间的元素
a[1...2] # => [2]   取下标为 1 到下标为 2之前的元素
```

图 3-19 取出部分元素的示例

修改元素值用下面的方法，给数据元素赋值。

```
b = [4,5,6]
b[0] = 2
#数组变为[2,5,6]
```

3.3.2 修改指定范围的元素内容

和引用一样，也可以指定范围来修改元素的内容。在赋值式的右端指定要替换的对象数组。

例如下面的例子，要替换从下标为 1 的元素开始的 0 个元素，结果是把右边的数据插入到下标为 1 的元素前面。

```
b[1,0] = [8,9]
#数组变为[2,8,9,5,6]
```

数组的长度也自动调整。下面的例子中，替换下标为 4 到下标为 4 之间的元素，所以只有下标为 4 的元素被置换。

```
b[4..4] = [1]
#数组变为[2,8,9,5,1]
```

下面的例子中，替换从下标为 3 到下标为 4 之前的元素，所以只有下标为 3 的元素被置换。

```
b[3...4] = [3]
#数组变为[2,8,9,3,1]
```

和 C 语言、Java 语言相比，Ruby 的数组具有以下特点。

- Ruby 的数组是对象，可以调用各种方法。
- 用[]访问数组实际上是方法调用。[] 是方法名，里面的值是参数。
- 变更数组元素实际上也是方法调用。[]= 是方法名，里面和右边的值是参数。

3.3.3 Ruby中的哈希处理

哈希是一个典型的集合，像数组一样被广泛使用。它用来表现对象和对象的对应关系。哈希表可以用哈希式来生成。图 3-20 中是定义星期的哈希式。

```
{ "星期天" => "Sunday",  
  "星期一" => "Monday",  
  "星期二" => "Tuesday",  
  "星期三" => "Wednesday",  
  "星期四" => "Thursday",  
  "星期五" => "Friday",  
  "星期六" => "Saturday" }
```

图 3-20 定义星期关系的 Hash 式

这种对应关系就像字典一样，所以哈希被称为字典也是基于这个原因。另外，由“星期天”可以联想到“Sunday”，哈希是这种联想关系的排列，所以也称为关联数组。

哈希表只是单向关联。所以上面的例子只是从汉语到英语的对应关系，并没有从英语到汉语的对应关系。相当于字典词条的部分称为键，翻译解释的部分称为值。

哈希表被称为关联数组还有一个原因。前面已经说明哈希是联想关系的排列。另外，数组可以看成是从整数（下标）到值的对应关系，可以把数组解释成整数键的关联数组。

其实，在 AWK 编程语言中数组和关联数组是一样的，下标不是整数的时候称为关联数组。所以 Ruby 也可以把哈希表称为下标不是整数的数组。

数组的元素是按照下标的整数顺序排列的，作为数据结构，哈希表中元素的顺序则是不定的。从 Ruby 1.9 开始，哈希表开始记录元素的顺序，即元素的插入顺序。从哈希表中取出元素的时候，顺序是由内部实现决定的，很难预测。这是内部算法的限制。哈希算法是很巧妙的算法，快速 ($O(1)$)² 实现从键到值的检索。使用哈希算法的集合称为哈希表。

2 $O(1)$ (O one) 是算法复杂度表示法定义的算法速度。算法复杂度和括号内的算式成正比。

除了键值可以不是整数，哈希表和数组没有太大的区别。假定 **h** 是指向哈希对象的变量，**key** 是键，就可以用下面的表达式取出键所对应的值。

```
h[key]
```

变更哈希表元素的时候，写法也是和数组一样的。

```
h["key"]="value"
```

3.3.4 支持循环的Enumerable

如果只有引用和变更元素这些基本功能的话，那么也就没有什么值得特别关注的。C 语言和 Java 语言也有数组。Ruby 中提供的方法可以更好地发挥集合的作用。

Ruby 中，集合的功能都定义在 **Enumerable** 这个 **Mix-in**³ 中了。从另一个角度来说，只要把 **Enumerable** 模块通过 **Mix-in** 继承进来，就可以使用集合对象的大量方法了。表 3-4 中列出了 **Enumerable** 提供的方法。

3 **Mix-in** 是抽象类之一。既具有单一继承的方法构成和优先顺序的明确性，又可以像多重继承一样继承多个类。

表3-4 **Enumerable** 提供的方法

方 法	说 明
<code>all?</code>	是否全部为真
<code>all?{ x ...}</code>	块对所有的元素是否都为真
<code>any?</code>	是否有元素为真
<code>any?{ x ...}</code>	块是否对有的元素为真
<code>collect{ x ...}</code>	块对各个元素处理结果的数组
<code>detect{ x ...}</code>	结果为真的第1个元素

<code>each_with_index{ x,i ...}</code>	按照下标顺序对各个元素处理
<code>entries</code>	元素的数组
<code>find{ x ...}</code>	块为真的第1个元素
<code>find_all{ x ...}</code>	块为真的元素数组
<code>grep(pattern)</code>	模式匹配的元素数组
<code>grep(pattern){ x ...}</code>	对模式匹配的元素数组进行块处理
<code>include?(x)</code>	和x相等的元素是否存在
<code>inject{ x,y ...}</code>	各元素的块处理结果
<code>inject(init){ x,y ...}</code>	各元素的块处理结果
<code>map { x ...}</code>	各元素的块处理结果的数组
<code>max</code>	最大的元素
<code>max { a,b ...}</code>	用块比较出的最大的元素
<code>max_by { x ...}</code>	用块变换后的最大的元素 (Ruby 1.9)
<code>member? (x)</code>	和x相等的元素是否存在
<code>min</code>	最小的元素
<code>min { a,b ...}</code>	用块比较出的最小的元素
<code>min_by { x ...}</code>	用块变换后的最小的元素 (Ruby 1.9)
<code>partition { x ...}</code>	用块把真假元素分开
<code>reject { x ...}</code>	块处理结果为假的元素集
<code>select { x ...}</code>	块处理结果为真的元素集
<code>sort</code>	对元素排序
<code>sort { a,b ...}</code>	用块处理来排序
<code>sort_by { x ...}</code>	用块处理变换后的结果来排序
<code>to_a</code>	元素的数组
<code>zip (a,...)</code>	连接各个集合到数组
<code>zip (a,...){ arr ...}</code>	连接各个集合后执行块

Enumerable 的意思是可数的，所以集成了对集合各元素进行循环处理的方法。这些方法大致可以分为以下 3 类：

- 循环
- 指定条件

- 排序与比较大小

3.3.5 用于循环的each 方法

循环处理的基本方法是 **each** 。下面的程序中是把 **col** 作为集合的例子。

```
col.each {|x| puts x}
```

把各个元素值赋给变量 **x** ，然后执行 **puts x** 。但是在 **Enumerable** 中并没有定义 **each** 方法。反之，**Enumerable** 中的所有方法都是在内部调用 **each** 方法来实现的。

只要是可以用 **each** 方法对各个元素进行循环处理的对象，如果用 **include** 包含了 **Enumerable** ，就可以使用表 3-4 中所表示的各种方法了。使用起来是非常方便的。

循环方法中最简单的是 **each_with_index** 方法。它的处理是在对元素循环时加上下标（图 3-21）⁴ 。

4 如果要执行图 3-21 中的 Ruby 程序，可以把图 3-21 中的内容保存在 **zu5.rb** 文件中，然后执行 Linux 的命令 **\$ruby zu5.rb** 。但是，必须要像 Fedora Core 4 那样，安装了 Ruby 的环境才可以。

```
["a","b","c"].each_with_index{|x,i|
printf "%d: %s\n", i, x}
# 输出
# 0: a
# 1: b
# 2: c
```

图 3-21 **each_with_index** 的示例

另外，最常用的方法之一是 **collect** 。**collect** 是对各个元素执行块处理，返回处理结果的数组。块的执行结果是块中最后一个表达式的值。比如下面的程序。

```
[1,2,3].collect{|x| x*2}  
# 结果  
# [2,4,6]
```

在 Ruby 的 `Enumerable` 中，像 `collect`、`select` 和 `detect` 等方法一样，名字以 `-ect` 结尾的方法有很多。这些方法名是继承自 `Smalltalk` 的。`collect` 的别名 `map` 也继承自 `Lisp` 编程语言中的方法名。

3.3.6 使用 `inject`、`zip` 和 `grep`

下面介绍几个不同于循环的方法。

`inject` 是用块来把各个元素结合起来。用语言很难解释清楚，请看图 3-22 中的示例。输出的结果是 120。

```
ary = [1,2,3,4,5]  
p ary.inject{|n,i| n * i} # => 120
```

图 3-22 `inject` 方法的示例

`inject` 首先把第 1 个和第 2 个元素传递给块。

```
1 * 2 => 2
```

然后把结果和第 3 个元素传递给块。

```
2 * 3 => 6
```

这样到最后，循环结果就变成了 $1*2*3*4*5$ 。用 `inject` 好像是往各元素之间注入了运算符，这就是 `inject`（注入）名字的由来吧。

zip 是从多个集合中并行取得元素的方法。具体请看下面的例子。

```
a = [1,2,3]
a = [2,4,6]
a.zip
# 结果
# [[1,2],[2,4],[3,6]]
```

没有块的 **zip** 调用，可以把复数个集合的第 **n** 个元素结合成一个数组返回。另外，如果有块的调用，那么参数的第 **n** 个元素都一起传递给块来处理（参见图 3-23）。这和图 3-24 中的处理基本上是相同的，因为没有生成多余的数组，效率就稍微高一点。

```
a = [1,2,3]
b = [2,4,6]
a.zip(b) {|x,y| printf "[%s,%s]\n",x,y}
# 输出
# [1,2]
# [2,4]
# [3,6]
```

图 3-23 zip 方法的示例，调用了块

```
a.zip(b).each{|x,y|printf
 "[%s,%s]\n",x,y}
```

图 3-24 zip 方法的示例，因为调用了 **each**，所以和图 3-22 中程序相比效率稍低

grep 方法可以对集合中的元素进行模式匹配。匹配用 **==** 运算符。和 **zip** 一样，如果指定了块，就不用生成数组，而是把各个匹配的元素传递给块来处理（参见图 3-25）。

```
a = ["foo","bar","baz"]
a.grep(/^b/)
# 结果（从b 开始的元素）
# ["bar", "baz"]

a.grep(/^b/) {|x|printf "%s\n",x}
```



```
# 输出
# bar
# baz
```

图 3-25 grep 方法的示例

3.3.7 用来指定条件的select 方法

对集合的各个元素进行块处理的是循环类型的方法。和它相对的，对各个元素进行块处理，用块处理的结果作为下个处理的判定条件的是条件指定型方法。

条件指定型方法中最常用的是 **select** 方法。**select** 方法把块处理结果为真的元素存放在数组中返回。这个方法的别名是 **find_all**。

```
a = [1,2,3,4]
a.select{|x| x%2 == 0}
# 结果（偶数）
# [2,4]
```

和 **select** 动作相反的方法是 **reject**。**reject** 方法返回块处理结果为假的元素的数组。

```
a = [1,2,3,4]
a.reject{|x| x%2 == 0}
# 结果（奇数）
# [1,3]
```

如果想找出第一个满足条件的元素，就可以用 **detect** 方法。它的别名是 **find**。

```
a = [1,2,3,4]
a.detect{|x| x%2 == 0}
# 结果(最初的偶数)
# 2
```

真假结果都需要的情况也不能说没有。这样一个八面玲珑的方法是 `partition`。`partition` 方法把真的和假的元素的数组作为块返回。

```
a = [1,2,3,4]
a.partition{|x| x%2 == 0}
# 结果
# [[2,4],[1,3]]
```

全部元素是否为真，或者至少有一个元素为真的判断方法是 `all?` 和 `any?`（参见图 3-26）。

```
a = [true,true]
b = [false,true]
a.all? # => true  (a 全部为真)
b.all? # => false (b 不是全部为真)
a.any? # => true  (a 有真的元素)
b.any? # => true  (b 有真的元素)
```

图 3-26 `all?` 和 `any?` 方法的使用示例

`all?` 和 `any?` 可以用块处理的结果作为判定条件。（参见图 3-27）

```
c = [1,2,3]
c.all?{|x|x%2==0} # => false
c.any?{|x|x%2==0} # => true
```

图 3-27 `all?` 和 `any?` 方法的使用示例，调用了块

3.3.8 排序与比较大小的方法

有一些方法是用来比较块执行结果的大小。

`min` 和 `max` 分别返回集合中的最小和最大的元素，如果调用了块，那么会把两个元素传递给块来比较大小。比较用 `<=>` 运算符， $a > b$ 的时候返回正整数， $a = b$ 时返回零， $a < b$ 时返回负整数。

```
ary = ["1", "11", "2", "6"]
p ary.max
# => "6" (字符串)
p ary.max{|a,b| a.to_i <=> b.to_i}
# => "11" (数值)
```

sort 方法把集合的元素按照从小到大的顺序排列。和 **min** 一样可以用块来比较大小（参见图 3-28）。

```
ary = ["1", "11", "2", "6"]
ary.sort{|a,b| a.to_i <=> b.to_i}
# 结果
# ["1", "2", "6", "11"]
```

图 3-28 sort 方法的示例

sort 的每次比较都要执行块处理，那么每次比较都要做整数化变换，这是有些浪费资源的。排序时要比较的次数很多，而比较次数也会随着元素个数的增加而增加。所以，如果先一次性把集合中全部元素都变换好的话，就会降低比较的开销，这种变换称为施瓦茨变换（Schwarzian Transform）。它是以 Perl 界著名人士 Randal Schwarz 的名字而命名的。

具体请看下面的例子（参见图 3-29）。

```
ary.map{|i|[i.to_i,i]}.sort.map{|j|j[-1]}
```

图 3-29 使用施瓦茨变换进行排序的示例

下面按照顺序来说明一下。从 **ary** 开始的前半部分

```
ary.map{|i|[i.to_i,i]}
```

是从元素计算用作比较对象的整数值（`i.to_i`），然后把它和元素一起放入数组。对数组排序，

```
.sort
```

最后从中取出原来数组的元素，就得到了排序结果。

```
.map{|j|j[-1]}
```

计算比较值的次数与元素个数是一样的，所以元素在很多时候排序的时间差别就会很大。因为这种写法是很难记住的，所以 Ruby 提供了施瓦茨变换的简单方法 `sort_by`。

```
ary = ["1", "11", "2", "6"]  
ary.sort_by{|x| x.to_i}  
# 结果  
# ["1", "2", "6", "11"]
```

和前面的施瓦茨变换相比，写法变得简单多了。和通常用块进行比较的 `sort` 方法相比也显得简洁。所以 `sort_by` 是既简单好用又效率高的方法。

Ruby 1.9 的开发版也提供了 `min_by` 和 `max_by` 方法，它们与 `min` 和 `max` 方法形式一样，可以用块来指定比较方法。

还有几个方法，不属于以上的循环、条件指定、排序及比较大小。比如有元素判定方法 `member?`（别名是 `include?`），返回所有元素的方法 `entries`（别名是 `to_a`）。

3.3.9 在类中包含（include）Enumerable 模块

如果有 `each` 方法，就可以通过包含 `Enumerable` 模块，来试一下 `Enumerable` 的各种方法。请看图 3-30 中的程序。最初先在类 `Dice`

（骰子）里指定了掷骰子的次数，然后包含 `Enumerable` 模块，最后把 3 以下的结果排除掉。

```
# (1-1) Dice 类定义
class Dice
  def initialize(n)
    @n = n
  end
  def each
    @n.times {
      yield rand(6)+1
    }
  end
end

# (1-2) Dice 对象的生成
# 生成要投掷10 次骰子
dice = Dice.new(10)
# 用each 方法来掷骰子
dice.each {|x| puts x}

# (1-3) 往 Dice 类中包含 Enumerable
# 增加功能
class Dice
  include Enumerable
end

# (1-4) 用继承的 reject 排除 3 以下的结果
# 已经生成的对象也会拥有新追加的功能
p dice.reject{|x| x<=3}
```

图 3-30 掷骰子的 `Dice` 类。定义类之后，通过包含 `Enumerable` 模块来增加方法

（1-1）中定义了 `Dice`（骰子）类。初始化方法 `initialize`（相当于 Java、C++ 的构造函数）中，给实例变量 `@n` 设置了投掷次数⁵。Ruby 中以 `@` 开始的变量是实例变量。

⁵ 实例变量是指各个对象中拥有各自独立值的变量。

`each` 方法中，用 `times` 方法做 `@n` 次循环。实际投掷骰子的结果是用随机数函数 `rand` 来生成的。`rand` 方法返回从 0 到指定参数之间的随机数。在图 3-30 的程序中，`rand(6)+1` 来生成骰子上从 1 到 6 的数。

生成的随机数用 **yield** 传递给块。调用 **yield** 可以使方法的执行暂时停止，转而来执行块的处理。块处理执行完了之后，程序再次从 **yield** 的下一行开始继续执行。

这里定义的 **Dice** 类，除了构造函数之外只有 **each** 这个方法。在（1-2）中，生成了 **Dice** 类对象，调用 **each** 方法来做投掷骰子的处理。但是如果只是表示 10 个数字的话，程序就没有趣味性了。

所以在（1-3）中，在 **Dice** 类中通过包含 **Enumerable** 模块来增加了功能。**Ruby** 中，可以用 **class** 语句给已经存在的类增加功能。这个例子中，本来一开始就可以在 **Dice** 类中包含 **Enumerable** 模块的，但为了介绍这个功能，我们就特意分开写了。

最后的（1-4）中，利用 **Enumerable** 的 **reject** 方法，把 3 以下的结果排除掉了。请注意已经生成的对象也可以用这个方法。

通过介绍 **Dice** 类的这个例子，我们是否掌握了块的调用方法和包含 **Enumerable** 模块的好处了呢？

3.3.10 列表内包表达式和块的区别

如上所述，我们用 **Ruby** 的块来对集合进行处理。但是在 **Python** 和 **Haskell** 编程语言中，用列表内包表达式（**List Comprehension**）功能来处理集合的场合比较多。这是处理数组的另一种方法。具体的定义方法如下。

```
[f(x) for x in col]
```

这个表达式的意思是，把 **col** 集合的各个元素赋值给 **x**，然后用 **f(x)** 来处理，最后返回结果的数组。如果用 **Ruby** 的 **Enumerable** 来实现的话，和我们前面学过的一样，程序如下所示。

```
col.collect{|x|f(x)}
```

列表内包表达式也可以像下面这样来处理满足特定条件的元素。

```
[f(x) for x in col if g(x)]
```

增加了 **if g(x)** 条件，返回的数组中就只包含满足 **g(x)** 条件的元素。用 Ruby 的话是像图 3-31a 那样。图 3-31b 则是 Ruby 1.9 的省略写法。

```
##(a)
col.select{|x|g(x)}.collect{|x|f(x)}
##(b)
col.select(&:g).collect(&:f)
```

图 3-31 相当于列表内包表达式 Ruby 写法

列表内包表达式写法是和自然语言（英语）很接近的，而且程序比用 Ruby 的 **Enumerable** 方法还更为紧凑，所以在 Python 编程语言的用户中非常受欢迎。Python 中也有和 **collect** 一样功能的 **map** 函数，但是因为列表内包表达式功能，甚至有人建议要废除这样的函数。

但是有人不喜欢列表内包表达式写法中的顺序。例如刚才的程序：

```
[f(x) for x in col if g(x)]
```

首先，把 **col** 的各个元素赋值给 **x**，然后执行 **g(x)**，**g(x)** 的结果为真的元素再执行 **f(x)**，最后把对 **g(x)** 为真的元素执行 **f(x)** 的结果作成列表，这个列表成为列表内包表达式的返回值。英文不好的日本人，可能会觉得很不容易理解。也许习惯了就好了，但总还是不够直观吧。

列表内包表达式是很深奥的，但是因为下面这些原因，Ruby 在将来也不会采用这个功能。

- 日本人（特别是我）读起来不太顺，魅力不够。

- 执行顺序相反，与 Ruby 的其他部分（基本是从左到右）不一致。
- 只能实现 `collect` 和 `select` 功能的组合，和列表内包表达式功能相比，块的应用领域更广（虽然程序有点冗长）。

因为这种功能是很有趣的，所以简单地介绍了一下。

从语法的外观到广泛的应用

在本文中提到了 Ruby 的块是继承了 MIT 开发的 CLU 编程语言的迭代器功能。CLU 的迭代器是使用 `for` 循环的内部迭代器。

Ruby 设计之初，对于块的实现考虑了好几种方法。比如用下面的 `using` 关键字来指定参数。

```
method using x
...
end
```

或者用 `do` 关键字明确地调用块。

```
do method using x
...
end
```

最终还是觉得现在的用法比较好。如果用 `for` 或者 `do` 这样的语句，那么就会妨碍循环抽象化以外的应用了。

设计时做出的“不用 `for` 语句”这个小小的决定，结果使得块得到了广泛的应用。想想这小小的“外观上的不同”带来的结果，真是不能小看啊。

语法相当于一般软件的用户交互界面，软件开发人员往往偏重于“它能做什么”这样功能性的一面，但从 Ruby 块的发展经历看来，“怎样实现”这样的用户界面实际上是非常重要的。

第 4 章 设计模式

4.1 设计模式 (1)

设计模式是个编程术语，它是指设计上经常反复使用的模式。这个词本来用于建筑界，表示各种各样的建筑物、街道的设计上共通的创意及构成的组合。即使在建筑界，这个词也是近些年来才开始使用的。设计模式这一思想好像起源于 Christopher Alexander 的 *The Timeless Way of Building*¹（牛津大学出版社，1979）一书。

1 其中文版为《建筑的永恒之道》，由知识产权出版社出版。——编者注

通常，建筑物的设计各不相同，另外加上用途、建筑条件等各种制约因素，一个设计是不能一成不变地套用到别的建筑物上的。建筑设计师只是通过重用积累的设计模式，试图缩短设计所花费的时间。

话说回到软件上来，重用的手法在软件行业比在建筑界中得到了更广泛的使用。一般是通过库的形式，各种软件共享处理过程、数据结构以及类等。实际上，Linux 等操作系统提供了数不清的各种库。

但是，只有库还不能达到充分地共享。软件设计中有些东西虽然不能以库的形式来把它独立出来，但是多个软件共通利用的东西是确实存在的。这样的东西只能称之为固定形式——模式。

比如，让我们来看一个最简单的模式吧。

```
for (int i=0; i<len; i++) {  
    ...  
}
```

这是称之为 **for** 循环的固定形式。在索引变量 **i** 从 **0** 到 **len** 变化的过程中进行相应的处理。有 C、C++ 或 Java 等编程经验的人，同样的代码恐怕见过成百上千遍了。但是，这个简单的处理并不能以库的形式来共享。这不是库，而应该称之为模式。

在软件设计中像这样的模式数不胜数。不过，设计人员自己很少能够意识到模式的存在，一般是在积累了很多经验之后，才几乎在无意识之中利用模式提高了软件开发的效率。

Erich Gamma 与几位合作者²一起，精选了软件设计中，特别是面向对象软件设计中反复出现的各种模式，比照着 Alexander 提倡的建筑上的概念，称之为“设计模式”。他们从自己及周围的开发经验中把模式总结出来并进行分类，出版了《设计模式》一书。《设计模式》中介绍了表 4-1 中列出的 23 种模式。

2 《设计模式》的作者中以 Erich Gamma 最为著名，其他 3 人 Richard Helm 、Ralph Johnson 和 John Vlissides 也是功不可没。他们 4 人被称为“四人帮”。

表4-1 《设计模式》中介绍的23种设计模式

模 式 名	内 容
Abstract Factory（抽象工厂）	用可配置的方法生成有关的对象群
Adapter（适配器）	变换对象的接口
Bridge（桥接）	分离类之间的实现
Builder（生成器）	分离复杂对象的生成过程
Chain of Responsibility（职责链）	用多个对象来处理请求
Command（命令）	把请求封装成对象
Composite（组合）	用树结构来构成对象
Decorator（装饰）	给对象动态增加新的功能
Facade（外观）	隐藏子系统的详细内容，提供统一的接口
Factory Method（工厂方法）	在父类只定义生成对象的接口，具体的生成过程由派生类来实现
Flyweight（享元）	以共享的方式提高大量小对象的实现效率
Interpreter（解释器）	语言解释器
Iterator（迭代器）	提供按顺序访问一组对象的方法
Mediator（中介者）	封装对象之间的相互作用
Memento（备忘录）	记录对象的内部状态
Observer（观察者）	把对象的状态变更通知给其他对象
Prototype（原型）	提供生成对象的原型
Proxy（代理）	提供控制对象访问的代理（容器）

Singleton（单件）	用来保证某个类的实例只有一个
State（状态）	把对象的内部状态独立出来，封装状态变化
Strategy（策略）	封装算法，使之具有可变换性
Template Method（模板方法）	父类定义框架，派生类具体实现其中一部分
Visitor（访问者）	对集合的元素进行操作

4.1.1 设计模式的价值和意义

Gamma 他们并没有发现新的模式。总结出来的 23 种设计模式也都是软件开发中早就存在并反复使用的模式，因此并不能说是 **Gamma** 他们的首创。但即使是这样，设计模式也还是得到了大家的关注。这到底是为什么呢？

首先，它明确了各种模式的效果和适用状况，给众多模式命名并进行分类，这本身就是非常有意义的。如果没有名字的话，即使使用了模式，程序员也大都没有明确意识到使用了模式。因为意识不到应该使用的模式，就会错过最合适的设计。

但是，他们最大的功绩并不在于把设计模式进行分类，而是明确了“软件中可以分类的设计模式”的存在。有了这样的认识之后，设计模式就不仅限于书中介绍的 23 个，人们开始发现和定义更多设计模式。

设计模式有了名字，人们就可以认识到它的存在。对于没有名字的东西，人们几乎不可能认识到它的存在，并对之进行讨论。这种不能用语言表达的知识我们称之为内隐知识。给这些在软件中经常反复出现的模式命名，使得这些本来只有经验丰富的程序员才能认识到的软件设计模式能被广泛认识和讨论。这样的知识称为形式知识。把迄今为止普通人难于学习和吸收的内隐知识变成形式知识的功绩是无与伦比的。

4.1.2 设计模式是程序抽象化的延伸

从软件设计进化的观点来看设计模式的话，可以把设计模式看成是软件抽象化的新工具。

到目前为止，把共通处理进行抽象化，结果产生了例程，把数据构造也包括在一起进行抽象化，结果产生了抽象数据类型，把抽象数据类型的共通部分再进行抽象，因而产生了继承这一工具。就像这样，软件设计总是在不断地导入新工具，以实现更高度的抽象化。

继承是把单一类的共通部分抽象化，以达到再利用的目的，但是，面向对象的系统很少是由单一的实现，几乎都是由很多类组合构成的。在这种类的组合中，就会有一些大同小异的模式出现在各种不同的系统中。这些模式是无法用类库来抽象的，为达到再利用的目的，设计模式这种形式是最有效果的。

有了设计模式这种方法，这些用类库所实现不了的类构成的模式就可以在各种各样的状况下得到应用。模式的分类是个很抽象的概念，的确是比单纯的例程或类库要难得多，但设计模式有可能带来极高的生产效率，这是其他方法所不能达到的。

类设计本质上是非常困难的，构思出来最优的一组类来达到目的，这件事并不是随便谁都能做到的。但是，一旦有了设计模式，只要把过去优秀的人们考虑出来的模式拿来应用一下，谁都可以做出优秀的设计。

4.1.3 Ruby中的设计模式

《设计模式》一书是用 C++ 和 Smalltalk 介绍模式实例的。看了那些例子，大家都会感觉到，绝大多数的模式用 Smalltalk 实现起来非常简单。这是为什么呢？

因为 Smalltalk 没有静态类型，所以也就不需要匹配类型的模板等机制，也不需要仅仅为满足类型要求而进行继承，这就是 Smalltalk 用起来很简单的原因。而且，由于语言本身的动态性质，有些模式根本不必要以模式的形式来抽象出来就可以得到简洁的实现，这也是 Smalltalk 显得简单的原因之一。

Ruby 在很多方面很像 Smalltalk，实现设计模式也毫无困难。一般说来，用 Ruby 来实现设计模式的场合，要比 C++ 简洁得多，有些模式用现成的库就足以表现。

下面以 Ruby 为中心，让我们来看几个在实际设计中应用设计模式的例子。

4.1.4 Singleton 模式

首先，让我们来看一下最简单的一个设计模式，Singleton（单件）模式。Singleton 模式用来保证某个类的实例只有一个。

为什么需要 Singleton 模式呢？比如作为其他对象的雏形而存在的对象（用于 Prototype 模式），以及系统全体只存在唯一的一个对象等，都要用到 Singleton 模式。

用 Ruby 实现 Singleton 模式的方法有几个，让我们按顺序来逐一说明。

使用 singleton 库的方法

Ruby 已经以库的形式实现了 Singleton 模式。如图 4-1 所示，使用 singleton 库的话，在任意的类里只要包含（include）上 Singleton 模块，那个类就变成了 Singleton 模式的对象。

```
require 'singleton'
class PrintSpooler
  include Singleton
  ...
end

PrintSpooler.instance.spool(document)
```

图 4-1 使用 singleton 库的 Ruby 代码

要想取得 Singleton 模式的类的对象，像图 4-1 最后一行那样，使用该类的 **instance** 方法。如果该类对象还没有生成，**instance** 方法会生成该类对象并返回。如果该类对象已经生成，**instance** 方法就返回既有对象。

使用类或模块

C++和 Java 是不能把类作为对象来使用的，与之不同的是，Smalltalk 或 Ruby 能把类也作为对象来处理。因此，在类或模块里定义一个方法就可以实现 Singleton 模式（参见图 4-2）。

```
class PrintSpooler
  def PrintSpooler::spool(doc)
    ...
  end
end

PrintSpooler.spool(document)
```

图 4-2 利用类定义来实现 Singleton 模式的代码

把一般的对象作为Singleton来使用

为了把一个类的对象限制成只有一个，并不一定需要对对象的一般生成方法加以限制。我们可以生成一个一般的对象，然后遵守绅士协定，不要再生成其他更多个对象，也就行了（参见图 4-3）。

```
class PrintSpooler
  def spool(doc)
    ...
  end
end

# 把唯一的对象赋值给一个固定变量
Spooler = PrintSpooler.new
Spooler.spool(document)
```

图 4-3 在编程上下点工夫来实现 Singleton 模式

使用对象和特异方法

其实还有不用类就可以实现的方法。Ruby 可以在对象生成之后再增加新的方法，这样我们就可以生成一个 Object 类的对象，然后给它追加必要的功能（参见图 4-4）。

```
Spooler = Object.new
def Spooler.spool(doc)
```

```
...  
end  
Spooler.spool(document)
```

图 4-4 利用特殊方法来实现 Singleton 模式的代码

这种使用特异方法的办法是很符合 Ruby 特征的。Ruby 自身的 `main`（n 最高层的 `self`）及 `ARGF`（虚拟文件，用来代表参数所指定的文件）等也都是用这种方法实现的。

4.1.5 Proxy 模式

Proxy（代理）模式是为某个对象提供代理对象的模式。为什么需要 Proxy 模式呢？

假设有个生成代价非常大的对象。如果在还不知道是否真正需要该对象的时候就事先生成它的话，可能会带来很大的浪费。但话虽这么说，不生成对象的话什么事也做不了。这时候代理对象就有用武之地了。

比如图像处理软件，它利用 Proxy 对象来处理嵌入图像，把嵌入图像的生成处理延迟到需要表示的瞬间才来进行。

Ruby 的库中也有使用 Proxy 模式的。比如 `tempfile` 库，它不用指定文件名就可以生成临时的工作文件（参见图 4-5）。

```
# 生成Tempfile  
f = Tempfile.new("foo")  
# 往Tempfile 输出  
f.print("foo\n")  
f.close  
# 再打开  
f.open  
# 读取数据  
p f.gets # => "foo\n"
```

图 4-5 采用 Proxy 模式的 `tempfile` 库的使用示例

Tempfile 类与实际负责文件输出的 **IO** 类没有继承关系，它的有关输入、输出处理的方法都通过 **Proxy** 委派到实际的 **IO** 类对象。因此，通过使用 **Tempfile** 类的对象，在任何有必要的时候也都可以使用相关的 **IO** 对象。

Proxy 模式也可以用 **Ruby** 的库来实现。使用 **delegate** 库就可以了。**delegate** 是委托的意思。**Tempfile** 类也是用 **delegate** 库来实现的（参见图 4-6）。

```
require 'delegate'

# 生成obj 的代理对象
proxy = SimpleDelegator.new(obj)

# 通过proxy 来调用obj 的some_method
Proxy.some_method
```

图 4-6 使用 **delegate** 库来实现 **Proxy** 模式的例子

看一下就知道，**delegate** 库的源代码是相当复杂的，但基本上只是把被调用的方法都委派到本来的对象那里去。这里使用的是 **Ruby** 的 **method_missing** 方法。

Ruby 中对对象 **A** 调用它所不知道的方法的时候，**A** 的 **method_missing** 方法就会被调用。传递给 **method_missing** 的参数是在原来调用方法的参数之前加上不存在的方法名。利用这一框架就可以很简单地实现 **Proxy** 模式（参见图 4-7）。

```
class Proxy
  def initialize(orig)
    @obj = orig
  end
  def method_missing(name, *args)
    @obj.send(name, *args)
  end
end

proxy = Proxy.new(obj)

# proxy.some_method 通过 proxy 来调用 obj 的 some_method
```


图 4-7 使用 `method_missing` 方法来实现 Proxy 模式的例子

怎么样，真的是非常简单吧。但是，这种实现方式也有不灵光的时候。Proxy 类固有的方法被调用的时候，是不会委派到 `method_missing` 方法的。也就是说，Proxy 类的父类 Object 类的方法是委派不了的。

如果这样的情况也要对应的话，就会稍微麻烦一些。实际上，`delegate` 库除去空行和注释以外还长达 114 行。比图 4-7 要复杂得多。

`delegate` 库使用起来虽然很简单，但方法委派的对象仅限于既有的对象。因此，在最开始举的字处理软件例子中，要想达到延迟图像生成的目的，直接使用 `delegate` 是不行的。我们可以像图 4-8 那样，从 `Delegator` 派生一个子类（`ImageProxy`）来达到这一目的。

```
require 'delegate'
class ImageProxy < Delegator

  # 传递延迟生成所需要的数据
  def initialize(data)
    @data = data
    @image = nil
  end

  # 下面的方法用来在访问时调查委派对象
  def __getobj__
    if @image == nil
      @image = LoadImage(@data)
    end
    @image
  end
end
```

图 4-8 Proxy 模式延迟对象生成的例子

`__getobj__` 方法是 `Delegator` 对象取得方法委派对象的方法。通过重写这个方法，`ImageProxy` 会在实际访问图像对象的时候才来生成图像对象。C++ 会用 `operator->` 或者 `operator*` 来代替 `__getobj__`。

4.1.6 Iterator模式

Iterator（迭代器）模式提供按顺序访问集合对象中各元素的方法。即使不知道对象的内部构造，也可以按顺序访问其中的每个元素。

Iterator 模式是为集合对象另外准备用来控制循环处理的对象，就像 C++ 或 Java 一样。我们称这个循环控制对象为 Iterator，也称为游标。

图 4-9 是 Iterator 模式的类构成图。调用集合对象（图 4-9 的 **Iterable**）的 **CreateIterator()** 方法，就会返回自己对应的 **Iterator** 对象。**Iterator** 对象会记住现在所指向的 **Iterable** 元素，调用 **Next()** 方法可以返回集合的下一个元素。要想知道集合中是否还有别的元素，可以调用 **IsDone()** 方法来确认。图 4-10 是利用 **Iterator** 模式的程序示例。**Iterator** 模式实现的是所谓外部迭代器的循环控制抽象化。

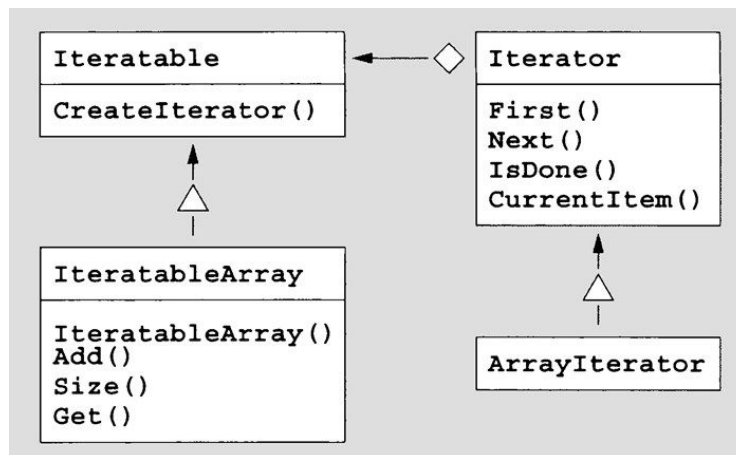


图 4-9 Java 版 Iterator 模式的类构成图

```
IterableArray array = CreateArray();
Iterator it = array.CreateIterator();
for (it.First(); !it.IsDone(); it.Next()){

System.out.println((String)it.CurrentItem());
}
```

图 4-10 Java 版外部迭代器的用法

而 Ruby 是用块来对集合的各元素进行循环处理的。作为设计模式，使用块进行循环的抽象化属于 Visitor（访问者）模式。但因为语言本身就支持这样的循环，所以也就不需要 Iterator 这样的对象了。这实在是太基本的东西了，也许都不应该称之为设计模式了。

4.1.7 外部与内部，哪一个更好

比较外部迭代器和内部迭代器，很难说哪一个更好。它们都有方便的一面，也都有不方便的另一面。比如，在没有闭包的语言中，要使用内部迭代器的话，就不能用块来实现，而是要传递给它函数指针，而且如果需要传递数据的话，就必须以参数的形式从外部明确传递过来，程序变得非常麻烦。

请看图 4-11。这是以 Ruby 和 C 为例编写的对哈希表进行循环的内部迭代器。Ruby 的内部迭代器是用块来实现的，代码看起来非常自然。但 C 是用函数指针来实现的，就比较难以理解。C 语言没有闭包，循环处理所需要的数据都要以参数的形式明确地从外面传递进去。说实话真是麻烦。如果把图 4-11 中省略的循环处理的实现部分也考虑进来的话，两者的差别是一目了然的。在没有闭包的语言中，实现内部迭代器是很不现实的。

```
# 用Ruby 对哈希表进行循环处理
h = Hash.new(...)
h.each {|k,v| ...}

// 用C 对哈希表进行循环处理
static int
each_func(st_data_t key, st_data_t value, st_data_t arg) {
    ...
    return ST_CONTINUE;
}
int main() {
    h = st_init_table(...)
    st_foreach(h, each_func, arg)
}
```

图 4-11 C 和 Ruby 的内部迭代器

因此，没有闭包的 C++ 或者 Java 语言通常使用另外准备的外部迭代器对象来实现循环控制。用 Java 来实现 Iterator 模式的程序大约 80 行左

右。图 4-12 仅是迭代器对象的实现部分的代码。程序变得如此长的原因主要是要解决类型的匹配问题。在 Ruby 中，对 Iterator 模式的类构成而言，当然是内部迭代器比较方便，但这并不表示不能使用外部迭代器。把刚才的 Java 版程序移植成 Ruby，程序还不到 50 行。图 4-13 是对应于图 4-12 的 Ruby 程序，Ruby 外部迭代器的使用方法如图 4-14 所示。

```
abstract class Iterator {
    abstract public void First();
    abstract public void Next();
    abstract public boolean IsDone();
    abstract public Object CurrentItem();
}

class ArrayIterator extends Iterator {
    private IteratableArray _array = null;
    private int _current = -1;
    public ArrayIterator(IteratableArray array) {
        _array = array;
        _current = 0;
    }
    public void First() {
        _current = 0;
    }
    public void Next() {
        ++_current;
    }
    public boolean IsDone() {
        return _current >= _array.Size();
    }
    public Object CurrentItem() {
        return array.Get(_current);
    }
}
```

图 4-12 Java 迭代器对象的例子

```
class ArrayIterator
  def initialize(array)
    @array = array
    @current = 0
  end
  def first()
    @current = 0
  end
end
```

```
def next()
  @current += 1
end
def is_done()
  return @current >= @array.size()
end
def current_item()
  return @array.get(@current)
end
end
```

图 4-13 Ruby 外部迭代器的例子

```
it = array.create_iterator();
it.first()
until it.is_done()
  puts it.current_item()
  it.next()
end
```

图 4-14 Ruby 版外部迭代器的用法

与 Java 版相比，你觉得怎么样？我喜欢更清楚易懂的 Ruby 版。

4.1.8 内部迭代器的缺陷

在有闭包的语言中，内部迭代器具有容易理解、容易实现以及自然封装等许多非常可取的性质。

但是，内部迭代器并不是完美无缺的。内部迭代器的缺陷是，由于不能同时进行多个循环，也就无法实现按顺序比较两个集合元素的处理。容易使用的东西也有它没有任何使用价值的领域。

比如，如果要从多个数组中把一个个元素取出来进行排列的话，就不能使用内部迭代器，而要写成图 4-15 的样子。

```
# 内部迭代器无法实现多个数组要素的并列处理
a = [1,2,3]
b = [9,8,7]
i=0
result = []
```

```

while i<3
  result.push(a[i])
  result.push(b[i])
end
result # =>[1,9,2,8,3,7]

# 用外部迭代器很简单
ia = a.create_iterator();
ib = b.create_iterator();
ia.first(); ib.first()
until ia.is_done() or ib.is_done()
  result.push(ia.next)
  result.push(ib.next)
end

```

图 4-15 内部迭代器的缺陷

但是 Ruby 也在改善之中。在 1.8.7 版本以后，几乎所有对块进行循环的方法，在没有块的时候，会返回像外部迭代器一样动作的 **Enumerator**。有了它，不用再特意准备外部迭代器，就可以把它作为外部迭代器来使用。

图 4-16 使用 **Enumerator** 实现了图 4-15 同样的处理。请注意，它比使用外部迭代器（图 4-15 的后半部分）的程序还要简单。

```

# 继续图4-15 的程序
result = []
ia = a.each
ib = b.each
loop do
  result.push(ia.next)
  result.push(ib.next)
end#

```

图 4-16 使用 **Enumerator** 进行外部迭代

实际上 **Enumerator** 对所有元素循环完了的时候会抛出 **StopIteration** 异常。**loop** 方法收到该异常后停止循环，而不需要像外部迭代器那样每次去问“还有别的元素吗？”，所以使用 **Enumerator** 的程序变得更简单。

4.1.9 外部迭代器的缺陷

那么，外部迭代器是不是就没有问题了呢？外部迭代器的缺陷在于迭代器（游标）对象需要引用集合对象的内部信息。为了按顺序访问集合对象的各个元素，迭代器对象需要访问集合的内部构造。这就破坏了隐蔽集合内部构造的封装性原则。因为集合类与迭代器类非常紧密地关联在一起，就需要特别注意它们内部构造的更新。

比如，C++使用 **friend** 修饰词来允许迭代器访问集合的内部构造。这就破坏了对对象的封装性原则。于是 C++只好使用 **friend** 来实现。

4.2 设计模式（2）

前一节学习了“设计模式是什么”。我们把设计上反复出现的模式称为设计模式。最简单的例子就是 **for** 循环。《设计模式》一书把设计模式进行了明确分类，极具参考价值。

上节讲述过 **Singleton**、**Proxy** 及 **Iterator** 各模式，本节再来考察个别的设计模式。下面按顺序来考察 **Prototype**、**Template Method** 和 **Observer** 这三个设计模式。

4.2.1 模式与动态语言的关系

《设计模式》一书介绍了 23 个设计模式。这些设计模式可以分为 3 大类：有关生成的模式（5 个），有关构造的模式（7 个）以及有关行为的模式（11 个）。如果把上节中三个模式进行这种分类的话，那么 **Singleton** 模式属于有关生成的模式，**Proxy** 模式属于有关构造的模式，而 **Iterator** 模式则属于有关行为的模式。

设计模式是从（面向对象）编程中经常出现的程序构造中抽象出来的，所以它与语言无关，可以适用于任何编程语言。《设计模式》的例题主要是用 C++写成的，其中也有用 **Smalltalk** 写的例题，但同样的原则对 **Java** 也完全适用。当然对 **Ruby** 也是完全一样的。

但是，**Ruby** 或 **Smalltalk** 这样的动态语言，与 C++或 **Java** 这样的静态语言相比，即使是同样的设计模式，使用方法也会略有不同。这次就着重从这一点来考察几个设计模式。

4.2.2 重复使用既存对象的Prototype模式

引用《设计模式》一书中的解释，Prototype（原型）模式“明确一个实例作为要生成对象种类原型，通过复制该实例来生成新的对象”。

《设计模式》中这一模式的例题使用的是迷宫生成类 **MazeFactory**。这一例子通过拥有生成墙、房间、门等对象的原型，不需要派生就可以生成各种各样的迷宫。但是，说实话，这个例题并没有让人真正感觉到原型的宝贵之处。

实际上，Prototype 模式本来并不太适用于 C++ 这样的静态语言。在动态语言中，Prototype 模式才能够真正发挥它的巨大威力。

通常在面向对象的语言中，首先准备好所谓的类，也就是对象的雏形，然后从类来生成实际的对象，这些做法都是理所当然的。在面向对象编程中，类是最为基本的存在。

但是，类真的是必需的吗？Smalltalk 的“亚种”Self 语言的设计人员就认为类并不是必需的。Self 中不存在所谓的类，基本操作并不是从类来生成对象，而是复制对象。

基本思想就是如此。

在需要新种类对象的时候，首先复制一个既存的对象，给复制的对象直接增加方法或实例变量等功能，生成最初的第一个新种类对象。如果该对象需要不止一个的话，那就从第一个雏型来复制，需要几个就复制几个。最初一个虽然是雏形，但它并不是所谓类这种人为生成的特别对象，而是一个普通的对象，只不过偶尔被用来复制而已。

雏形这个词对应的英语是 **Prototype**。像这样的不用类，而是用原型模式和复制方法的编程语言称为原型模式的编程语言。实际上，Prototype 模式不单是一种设计模式，也许称为一种编程范例才更为合适。

相对于类模式的编程，原型模式的编程的构成元素比较少，具有简单实现面向对象功能设计的倾向。因此，最近有越来越多的规格较小的编程语言采用这种模式。比如，大多数 Web 浏览器中嵌入的

JavaScript 的面向对象功能就是原型模式的。最近，受到一部分人关注的 Io 语言¹ 也是原型模式的。

1 关于原型模式的面向对象编程语言 Io，请参阅 <http://www.iolanguage.com/>。关于语言的基础，请参阅笔者在 *Rubyist Magazine* 上的投稿 (<http://jp.rubyist.net/magazine/?0010-Legwork>)。

4.2.3 亲身体验Io语言

只讲理论还是难以得到具体的印象，那就让我们边看实际程序，边来亲身体验原型模式的编程吧。虽然用 Ruby 也可以进行原型模式的编程，但这次我们使用更为彻底的 Io 语言（参见图 4-17）。

```
// 复制Object 1
Dog := Object clone

// 把sit 方法教给雏形Dog 2
Dog sit := method("I'm sitting.\n" print)

// Dog 是狗，所以会坐 3
Dog sit

// 从雏形Dog 生成新的myDog 4
myDog := Dog clone
```

图 4-17 使用 Io 语言的原型模式的编程示例

让我们来仔细看看图 4-17 吧。这是描述狗的简单例子。虽然没有实用性，但从中应该能体会到原型模式的气氛。

1 的部分生成新的对象，赋值给名为 **Dog** 的变量。请注意，这里出现的 **Object** 和 **Dog** 都不是类。在 Io 语言中，**Object** 只是有代表性的对象，除最基本的以外其他什么都不知道。以它为基础可以生成各种雏形。这里调用 **clone** 方法来复制一个基础对象，并给它起个名字，叫做 **Dog**。刚刚复制出来的 **Dog** 对象跟 **Object** 是一样的，没有狗的任何功能。

只是 **Object** 的话，什么功能都没有，是没有任何使用价值的，让我们来教给它狗的功能吧。2 部分中先是给它定一个“坐下”的 **sit** 方

法。把一个 **method** 对象赋值给 **Dog** 对象的 **sit** 属性，就给 **Dog** 对象追加了一个方法。

调用 **sit** 方法就等于调用 `"I'm sitting\n"print`，显示 **I'm sitting.**。这一部分相当于调用字符串对象的 **print** 方法，从中可以体会到彻底的面向对象编程。这个例子中仅仅增加了一个 **sit** 方法，实际上根据需要想追加几个方法就可以追加几个。

你看，**Dog** 对象就是这样实现的。再强调一遍，其中作为雏形的是狗对象，而不是类。因此，3 部分中对 **Dog** 对象调用 **sit** 方法的时候，结果与其他狗一样显示出 **I'm sitting.**。

在像 **Ruby** 这样类模式的语言中，作为对象雏形的类拥有与对象完全不同的方法（类的方法），而与之相对的是，原型模式的语言中根本就没有类的存在，雏形与基于它生成的对象是完全一样的。

4 部分使用 **clone** 方法基于雏形生成新的狗对象。这里请注意，不管是生成新的雏形，还是从雏形生成新的对象，都是仅仅使用 **clone** 方法实现的。在类模式的语言中，用子类化和实例化这两个不同的概念实现的程序，这里仅仅用 **clone** 这样一个简单的程序就实现了，真让人感动。

当然，简单并不都是一切，原型模式有原型模式的优点，类模式也有类模式的优点，因为原型模式的语言还不太广为人知，这里特意选它作为例子，就是为了让大家体会一下它的单纯。

4.2.4 Ruby中的原型

基本上讲 **Ruby** 是类模式的语言，但也拥有支持原型模式编程的功能，具体来说有以下 3 种功能。

1. 复制对象的 **clone** 方法。
2. 给个别对象增加方法的特异方法功能。
3. 给个别对象增加一组功能的 **extend** 方法。

图 4-18 是用 **Ruby** 重写的图 4-17 的程序。

```
// 生成雏形对象
object = Object.new
// 复制object
dog = object.clone
// 给雏形dog 增加sit 方法
def dog.sit
  print "I'm sitting\n"
end

// dog 是狗，会sit
dog.sit

// 从雏形dog 生成新的myDog
myDog = dog.clone
```

图 4-18 Ruby 的原型模式编程，这是用 Ruby 重写的图 4-17 的内容

因为 Ruby 中没有像 Io 语言中 **Object** 这样一个对象原型，所以作为准备，程序一开始（`object = Object.new`）先是生成一个 **Object** 类的实例。在这以后，除了语法上细微的区别之外，差不多是把 Io 程序照搬过来的。

从中我们可以看到动态语言中 **Prototype** 模式这种令人吃惊的力量。这在必须明确对象类型的静态语言中是实现不了的。因为在静态语言中没有原型编程，是不可能给复制的对象增加新方法的。

4.2.5 编写抽象算法的Template Method模式

接着来看 **Template Method**（模板方法）吧。还是引用《设计模式》中的解释，**Template Method** 模式是：“在父类的一个方法中定义算法的框架，其中几个步骤的具体内容则留给子类来实现。使用 **Template Method** 模式，可以在不改变算法构造的前提下，在子类中定义算法的一些步骤。”

这实际上是面向对象编程中使用继承的一般技巧。

作为例子，让我们来看一下 Ruby 的 `p` 方法吧。`p` 方法是程序调试中用来显示对象内容的方法。显示调试信息的算法主要是：

1. 把对象的调试用输出信息转换成字符串；
2. 调用 `puts` 方法输出。

这就是调试输出的算法，因为实在是太简单了，称之为算法简直有些过分。

但实际上令人意外的是，为了输出调试信息，分别为各种对象定义调试用输出字符串是非常困难的。针对各种不同的类都要分别进行不同处理，这就很困难了，每次增加新类的时候，为支持新类也都需要做大量的工作。

这时候使用 **Template Method** 模式，问题就变得简单了。使用 **Template Method** 模式，输出调试信息的 `p` 方法会变成如下的代码，简单得出乎意料。

```
def p(obj)
  puts obj.inspect
end
```

这个简单的方法只是把算法的 1 和 2 原封不动地换成了程序语言。在这个定义中，各种对象中准备调试信息的具体处理是由该对象的 `inspect` 方法来实现的。在定义新类的时候，只要给它定义了合适的 `inspect` 方法，就可以在任何时候使用 `p` 方法来输出适当的调试信息。

在父类中定义抽象化的算法，调用隐藏了实现细节的方法，然后在子类中实现具体的细节，这就是 **Template Method** 模式。

4.2.6 用Ruby来尝试Template Method

Ruby 的类库中最大限度灵活运用 **Template Method** 模式的部分，应该说是 **Enumerable** 模块和 **Comparable** 模块了。

Enumerable 模块中实现循环的 `each` 方法采用了 **Template Method** 模式。表 4-2 是 **Enumerable** 模块的方法一览。

表4-2 Enumerable 提供的方法

方法名	功 能
all?	是否所有元素都为真
all?{ x ...}	块是否对所有元素都为真
any?	是否至少有一个元素为真
any?{ x ...}	块是否对至少有一个元素为真
collect{ x ...}	对各元素进行块中的计算，返回结果的数组
detect{ x ...}	返回使块为真的第1个元素
each_with_index{ x,i ...}	对各元素和下标进行块中的计算
entries	返回元素的数组
find{ x ...}	返回使块为真的第1个元素
find_all{ x ...}	返回使块为真的所有元素的数组
grep(pattern)	返回匹配检索模式的所有元素的数组
grep(pattern){ x ...}	对匹配检索模式的所有元素进行块中的计算
include?(x)	是否有元素与x相等
inject{ x,y ...}	返回对各元素进行块中的计算的结果
inject(init){ x,y ...}	返回对各元素进行块中的计算的结果
map{ x ...}	对各元素进行块中的计算，返回结果的数组
max	返回最大的元素
max{ a,b ...}	使用块中的比较方法，返回最大的元素
max_by{ x ...}	对各元素进行块中的变换，返回结果最大的元素（Ruby 1.9）
member?(x)	是否有元素与x相等
min	返回最小的元素
min{ a,b ...}	使用块中的比较方法，返回最小的元素
min_by{ x ...}	对各元素进行块中的变换，返回结果最小的元素（Ruby 1.9）
partition{ x ...}	把使块为真的元素和使块为假的元素分离开
reject{ x ...}	返回使块为假的元素的数组
select{ x ...}	返回使块为真的元素的数组
sort	对元素排序
sort{ a,b ...}	使用块中的比较方法，对元素排序
sort_by{ x ...}	对各元素进行块中的变换，按照变换结果对元素进行排序

to_a	返回元素的数组
zip(a,...)	返回各集合串接后的数组
zip(a,...){ arr ...}	对集合进行串接，然后进行块中的计算

这些方法的定义都是仅仅依赖于 **each** 方法。因此，在用户定义的类中，只要定义了 **each** 方法，一旦把 **Enumerable** 模块包含（**include**）进来，就都可以使用表中的 32 个方法了。

Enumerable 模块实际上是用 C 编写的，用 **Ruby** 也可以简单地定义同样的模块，那就让我们边看 **Ruby** 的定义，边来考虑一下如何更好地使用 **Template Method** 模式²。

2 在 **Ruby** 早期的版本里，**Enumerable** 模块是用 **Ruby** 定义的。后来考虑列性能及库的使用方便性，又用 C 重新编写了。

其中一个实现起来最为简单的方法，应该是收集所有元素的 **entries** 方法了。**entries** 方法的实现代码如图 4-19 所示。看一下图 4-19 就能明白，处理内容是很简单的。

```
def entries
  result = []
  self.each {|elem|
    result << elem
  }
  return result
end
```

图 4-19 用 **Ruby** 实现的 **entries** 方法

- 1. 生成数组。
- 2. 用 **each** 取出每个元素。
- 3. 把元素追加到数组里。
- 4. 最后返回数组。

从中可以看出，这个方法只是定义了自己处理的框架，对应于每个对象的处理则由 **each** 方法来提供。

像 **Template Method** 模式的这种使用方法，在 **Comparable** 模块中也是一样的。

Comparable 模块利用基本的比较大小方法 `<=>`，提供各种比较演算。`<=>`方法把自身与参数相比较，如果自身较大，则返回正整数；若二者相等，则返回 0；若自身较小，则返回负整数。以这个方法为基础，**Comparable** 模块提供了 `==`、`>`、`>=`、`<`、`<=` 以及 `between?` 共 6 种比较运算。

作为 **Comparable** 提供的比较运算的代表，我们来看一看 `>` 方法的实现吧（参见图 4-20）。实际上 `>` 方法还要加上错误处理，但基本处理如图 4-20 所示。

```
def >(other)
  cmp = self <=> other
  if cmp > 0
    return true
  else
    return false
  end
end
```

图 4-20 用 Ruby 实现的 `>` 方法

像这样使用 **Template** 模式，可以不涉及各种数据结构细节，而只在抽象的水平上编写算法的程序。也就是说，算法是在抽象水平很高的状态下表述的，同样的代码能够适用于各种各样的情况。

这样避免了代码的重复，从 **DRY** 原则的观点来看³，也是很优秀的。

³ 所谓 **DRY**（Don't Repeat Yourself）原则，是指彻底避免重复。在几乎所有领域，这都是一个提高软件开发效率和可靠性的有效原则。希望大家时刻牢记在心。

4.2.7 动态语言与 **Template Method** 模式

一般 Template Method 模式与继承往往是成对讨论的，但像 **Enumerable** 那样，只需要包含 (**include**) 进来，不管继承关系如何，即可以向任何类里追加功能，这一点很有魅力。本来，Ruby 的 **include** 就是一种受限的多重继承，这没有什么不可思议的。

Template Method 模式的这种优秀性质与语言是不是静态没有关系。像 Java 那种含有静态类型，而且不允许多重继承的语言，必须强制性地拥有继承关系。所以，像 **Enumerable** 这样在各种各样的类中都能利用的算法集，使用 Template Method 模式很难实现 (**interface** 与委托的组合也不是不可能)，但这不是静态语言的问题。

但是，像 Io 与 Ruby 这种也善于用原型模式编程的语言，往前进化了一步，可以往特定对象里追加算法集。图 4-21 表示往特定对象里追加 **Enumerable** 功能，虽然这个例子有点牵强。

```
# 生成一个对象
dice = Object.new
# 定义each 方法
def dice.each

  # 首先掷10 回骰子
  10.times do
    yield rand(6)+1
  end
end
# 往dice 里追加Enumerable 的功能
dice.extend(Enumerable)

# 用继承的reject 方法排除3 以下的数
p dice.reject{|x| x<=3}
```

图 4-21 往特定对象里追加 **Enumerable** 功能的示例

尽管哪儿也没定义类，但使用 **extend**，骰子对象中就能够利用 **Enumerable** 模块的功能了。用 **extend** 及特异方法往特定对象里追加功能的做法，也能够用来实现 Singleton 模式。

4.2.8 避免高度依赖性的Observer模式

Observer（观察者）模式是当某个对象的状态发生变化时，依存于该状态的全部对象都自动得到通知，而且为了让它们都得到更新，定义了对象间一对多的依存关系。

这是控制类与类之间依存关系的一种模式。举一个例子，想想微软的 **Excel** 软件吧。以表中的数据为基础表示图形的时候，编辑了表中的数据之后，自然希望图形的内容也跟着变化。或者，从同一组数据，也经常想同时看到直方图和扇形图等多种图形。

能够实现这一要求的最简单的方法，应该是在表编辑功能里附加更新图形显示的处理。但是这样做的话，附加的是与表编辑在本质上不同的处理手段，使事情复杂化，更重要的是，当想要再利用表编辑功能时，还要牵连到不一定必要的图形显示功能。表编辑功能与图形显示功能之间的这种关系称为高度依赖性。

一般地说，高度依赖性不好。从本质上讲，软件是个复杂的东西，为了控制复杂性，有效的方法是将整体分割成几个相互独立的部分进行开发。但是，有了高度依赖性，就不能将组成程序的“零件”（类以及子程序）进行分解，一个一个的“零件”会很大，结果复杂性就很难控制。

Observer 模式是一种避免这种高度依赖性的手段。构成观察者模式的有两个对象，一个称为 **Observer**（观察者），接受变更通知；另一个称为 **Subject**（对象）或 **Observable**（被观察者），发出变更通知。

说说刚才的表编辑的例子，表数据就是 **Subject**，图形就是 **Observer**。从观察者与被观察者这两个名字上，被观察者让人得到被动的印象，在实际处理中，被观察者会发出通知“我已经变化了哦”。

4.2.9 Observable 模块

Ruby 中为实现 **Observer** 模式提供了名为 **observer** 的库。**observer** 库提供 **Observer** 模块。**Observer** 模块的 API 请参见表 4-3。实际的库使用如图 4-22 所示。

表4-3 Observable 模块的API（应用编程接口）

方法名	功 能
-----	-----

add_observer(observer)	增加观察者
delete_observer(observer)	删除特定观察者
delete_observers	删除观察者
count_observers	观察者的数目
changed(state = true)	设置更新标志为真
changed?	检查更新标志
notify_observers(*arg)	通知更新。如果更新标志为真，调用观察者带参数 args 的方法

```
require "observer"

# 更新通知者 (Observable)
# 这个类每秒发送1 次更新通知
class Tick

  include Observable
  def tick
    loop do
      now = Time.now
      changed
      notify_observers(now.hour, now.min, now.sec)
      sleep 1.0 - Time.now.usec / 1000000.0
    end
  end
end

# 观察者 (Observer)
# 依照通知，表示现在时刻的类（文字版）
class TextClock

  def update(h,m,s)
    printf "\e[8D%02d:%02d:%02d", h, m, s
    STDOUT.flush
  end
end

tick = Tick.new
tick.add_observer(TextClock.new)
tick.tick
```

图 4-22 使用 observer 库的 Ruby 程序，观察者与被观察者不相互依存

解释一下图 4-22 中的程序。首先是 `require observer` 库。利用库的时候，这是必须写的。

然后是定义被观察者类 `Tick`。注释中也写道，该类每秒发送 1 次更新通知。它是相当于时钟的类。`Tick` 的发音，好像是时钟的滴答滴答声。`Tick` 是被观察者，所以要将 `Observable` 模块包含进来。仅一句话就能让任意类成为被观察者，这正是 Ruby 的威力。

`tick` 方法是主循环。有了这个处理，每隔 1 秒，循环就发出更新通知。虽然仅仅睡眠 1 秒，但为了保证能在整秒发出更新通知，便以微秒为单位进行了补正（`sleep 1.0 - Time...` 的部分）。

实际的更新通知只是调用 `changed` 方法设置更新标志，然后用 `notify_observers` 方法通知观察者。它们都写在 `loop`（循环）内。

虽然在这种每次肯定都要定期发出更新通知的情况，把 `changed` 与 `notify_observers` 分离开来没有意义，但是考虑到会有频繁变化、每次更新处理的花费都比较大的情况，还是将二者分离开了。比如刚才的表编辑的例子中，与其在每次细微的变化后都要更新图形，不如在键盘输入告一段落时再集中更新图形，应该更有效率。

后半部分的 `TextClock` 类是观察者类。依照 `Tick` 发送的通知，在控制画面上显示现在时刻。`TextClock` 类不是特定类的子类或者别的什么，只是拥有被更新通知调用的 `update` 方法。`update` 方法接受 `Tick` 类 `notify_observers` 方法传过来的时、分、秒三个整数参数。

实际显示用了 ANSI 的转义字符（`printf` 以下的部分）。用 `ESC[8D` 将光标移到行首，后面显示时刻。为避免缓冲问题，每次都调用 `STDOUT.flush`。

定义了 `Tick`（被观察者）与 `TextClock`（观察者）两个类之后就简单了。先生成 `Tick` 类的对象（图 4-22 倒数第 3 行的部分），然后使之与 `TextClock` 类相关联，最后启动 `Tick` 类的主循环（图 4-22 的末尾部分），就这么多。

执行图 4-22 的程序，控制画面上就会显示出一个数字时钟。因为是无限循环，想要停止时，请按下 **Ctrl+C**。

这次只做了一个观察 **Tick** 类的对象 **TextClock**，如果愿意，可以添加任意多个观察者。比如，对同一个 **Tick**，不光能添加文字时钟，还可以添加图形时钟。

这个程序最应该注意的一点是，**Tick** 类与 **TextClock** 之间的关联，只用一行（图 4-22 倒数第 2 行）就完成了。**Tick** 类与 **TextClock** 类之间，只有“更新以后，调用 **update** 方法”以及“在 **update** 方法中，传递时、分、秒”这种简单的约定，不存在别的关系。只要是遵守相同约定的类，都可以简单地进行交换。

可以看出，使用 **Observer** 模式，显然能够降低相互依赖性。既可以将观察者类做成零部件，又可以根据需要更换被观察者（比如测试用的假程序）。这个性质对于提高软件的开发效率和测试效率，都是很有用的。

4.2.10 **Observer**模式与动态语言

动态语言的性质在 **Observer** 模式中也很有用。由于 **Ruby** 的动态性质，**observer** 库具有以下几方面的灵活性。

1. 观察者类不必是特定类的子类。
2. 观察者类不必实现特定的接口（本来在 **Ruby** 中也没有接口）。
3. 观察者类的更新方法名可以自由决定（**Ruby 1.9** 的功能）。
4. 观察者类更新方法的参数可以自由决定。
5. 被观察者类不必是特定类的子类。
6. 对被观察者类的要求，只是将 **Observable** 模块包含进来。

我想 **Java** 那种静态语言也具有与 **Ruby** 的 **observer** 库相同功能的库。事实上，有几种 **DI** 容器（**Dependency Injection Container**）框架，也具有与 **observer** 库相类似的处理。

但是，如果编码太繁杂了，或者需要用 XML 文件代替 Java 来描述类之间关联的话，我认为就没有 Ruby 这么好用了。

* * *

本节从与动态语言相关联的观点解释了设计模式中的 **Prototype** 模式、**Template Method** 模式和 **Observer** 模式。作为对设计模式的总结，下面看一看设计模式与软件开放—封闭原则（**Open-Close principle**）。

4.3 设计模式（3）

很久以前，技术人员将计算机的机械部分称为硬件，这是计算机所有实体部件和设备的统称。与此相对应，没有实体的程序被称为软件。如今硬件和软件作为计算机关联用语已经固定下来了，但当初却是技术人员之间的俗语。

说起软件，会让人想起“柔软灵活”，但事实上，缺乏灵活性的东西有很多。程序规模小的时候，还能够简单地更改，让人觉得有灵活性。但对于那些大规模商用软件，各部分依存关系很紧密，改动一个地方就会对别的地方有影响，总是不能随心所欲地更改。

4.3.1 软件开发的悲剧

在软件开发过程中，会遇到各种各样的问题，原因归结起来主要有两个方面，一个是复杂性，一个是变化性。

软件的规模越大，各个部分之间的牵连越复杂，更改也就越难。如果软件单纯而且规模小，更改还相对容易。随着计算功能的提高，交给计算机的任务规模也越来越大。几乎所有的软件，都随着用户需求的提高而得以扩展，变得越来越复杂。

如果只是增加软件功能，也不会引起那么多的问题。但是，在软件开发过程中，需求变更几乎是不可避免的。在洽谈时，即便已经同意了画在纸上的软件模型，可一旦见到了程序，“还是感觉不对劲，希望再改一改”，很多用户都会这么说。我自己作为一个多年的职业程序员，对于用户想到哪儿就是哪儿的作法，也经常发牢骚。

话虽这样说，前几天，我委托同事写了一个程序，尽管事前已经同意了需求，但看了实际程序以后，还是忍不住说：“与想象的稍微不同，能这样改一改吗？”自己竟然也跟那些被我发过牢骚的用户完全一样了，唉，只能感叹人是多么地自私任性。

先不说这些了，尽管软件越来越复杂，更改所需要花费越来越大，但用户要求却越来越多样化，对软件的变更要求也越来越频繁。长此以往，软件开发肯定要在什么地方失败。

4.3.2 开放—封闭原则

面对以上情况，有用的原则是开放—封闭原则（**open-closed principle**）。开放—封闭原则是 Eiffel 语言的设计者 Bertrand Meyer 在其著作《面向对象的软件构造》中介绍的原则。其定义如下，非常简单。

对模块扩展必须开放（Open），对修改必须封闭（Closed）。

所谓“对模块扩展必须开放”，是指模块可以扩展。比如，如果数据结构能够追加新的字段，或是能够追加新的功能，就可以称模块是开放的。某一模块会被用到什么地方，不可能完全预测。为了应对将来的需要，对于扩展必须是开放的。

所谓“对修改必须封闭”，是指某一模块被别的模块引用时的要求。必须做成这个样子：即使被引用一方的实现细节发生变化，也不会带来问题。

也就是说，即使某一模块的内部结构改变了，对外接口也应当是不变的。如果对外接口不能保持不变，模块就不能稳定使用。使用不稳定的模块，别的模块也必须时常跟着改变，软件的复杂性和维护成本都要增加。

把 **open-closed principle** 译作“开放—封闭原则”，感觉有点生硬。不管怎样，一次又一次重复“开放—封闭原则”都显得有点冗长，以下简称为 **OCP**。

4.3.3 面向对象的情况

既要开放，又要封闭，这看起来互相矛盾。但是面向对象编程语言能够很彻底地消除这个矛盾。

请看图 4-23。这个程序里有 3 种箱子（普通的箱子、上了锁的箱子及扎着彩带的箱子），要根据箱子的种类来打开箱子。

可以看出，这是一个很漂亮的程序，用最少的代码就能实现想做的事。如果想让这个程序对应新种类的箱子，只需要生成新的箱子对象，为这个箱子定义 **open** 方法就行了。所以，可以说，这个程序对于修改而言是封闭的¹。

1 这句话应该是，这个程序对于扩展而言是开放的。——译者注

```
面向对象的方法（OCP）
# 变量box1、box2 和box3 分别是3 种箱子

def box1.open()
    puts("打开箱子")
end

def box2.open()
    puts("解锁，打开箱子")
end

def box3.open()
    puts("解开彩带，打开箱子")
end

box1.open() # 显示“打开箱子”
box2.open() # 显示“解锁，打开箱子”
box3.open() # 显示“解开彩带，打开箱子”
```

图 4-23 3 种打开箱子的代码，面向对象的情况，满足 OCP 原则

3 种箱子是各不相同的对象，打开箱子只要调用

```
box.open()
```

就行了，对哪个箱子都一样。即使将来箱子种类增加了，只要那个对象有 **open** 方法，就可以同样处理。结果，即使追加了箱子，也不用更改现有代码。所以，这个程序对于修改而言是封闭的。

OCP 初看起来似乎是自相矛盾的，但如果使用面向对象语言来编写程序的话，就完全能够达到它的要求。

4.3.4 非面向对象的情况

那么，让我们来看看用非面向对象语言编写程序来进行相同处理的情况（参见图 4-24）。在这个程序中，将来箱子种类增加时，需要更改 **box_open** 子程序。这次与箱子关联的子程序只有 **box_open**，如果有多种子程序，单纯考虑各种组合，就会有以下这么多。

```
def box_open(box)

# box_type(box) 假定由子程序能知道箱子种类
  if box_type(box) == "plain"
    puts("打开箱子")
  elsif box_type(box) == "lock"
    puts("解锁，打开箱子")
  elsif box_type(box) == "ribbon"
    puts("解开彩带，打开箱子")
  else
    puts("不知道打开方法")
  end
end

# 变量box1, box2, box3 分别是3 种箱子

box_open(box1)    # 显示“打开箱子”
box_open(box2)    # 显示“解锁，打开箱子”
box_open(box3)    # 显示“解开彩带，打开箱子”
```

图 4-24 3 种打开箱子的代码，非面向对象的情况，不满足 OCP 原则

子程序的种类 × 箱子的种类

箱子种类增加了，组合的种类也会急剧增加。这样就不能随便增加箱子种类，对于扩展而言，不能说是“开放”的。

反之，根据箱子种类的不同而调用不同的子程序，会怎样呢？（参见图 4-25）这样增加箱子种类，对各种箱子定义了独立的子程序，因此可以说对于修改而言是封闭的。但是，对于调用子程序的那一侧而言，需要考虑到箱子的种类。因此，很难说它对于扩展而言是开放的。

```
def plain_box_open(box)
  puts("打开箱子")
end
def locked_box_open(box)
  puts("解锁，打开箱子")
end
def ribbon_box_open(box)
  puts("解开彩带，打开箱子")
end

plain_box_open(box1)
locked_box_open(box2)
ribbon_box_open(box3)
```

图 4-25 打开 3 种箱子的代码，别的非面向对象型代码

使用面向对象语言，功能使用方可以不必知道功能提供方各种类的详细内容，而只需要着眼于它们具有什么样的接口就可以。功能扩展以后，由于多态，扩展后的功能能够自动使用，使用方只要知道接口就行了。功能提供方提供了新功能，或是内部有了更改，功能使用方都不用做任何更改。也就是说，从功能的使用方来看，模块对于修改是封闭的。

另一方面，功能的提供方只要生成与既存对象具有相同接口的对象，任何时候都能追加新的功能。也就是说，对于功能扩展而言是开放的。这种情况下的接口，对于静态语言来说，就是相同的类型；而对于动态语言来说，就是有没有相同的方法（换个说法就是 **Duck Typing**²）。

2 所谓 **Duck Typing**，是指这样一种思考方法：如果某个东西，其行为跟鸭子一样，那么不管它是不是鸭子，都将它看作鸭子。这种想法不考虑某种对象所属的类，而只关心它具有什么

样的行为（具有哪些方法）。

很多面向对象语言，通过使用继承，只要添加一个子类，就可以往模块（类）里随时追加功能。因为有继承而允许功能的追加（对功能扩展而言是开放的），因为有多态而维持模块接口的稳定性（对修改而言是封闭的），开放和封闭能同时实现。

从实用主义的观点看，面向对象的精髓就在于对 **OCP** 的实践。至于把对象看做物体理解起来比较容易，能够建立现实世界的模型等，这些都只不过是些锦上添花的东西。

“数据与函数没有一体化，所以不是面向对象”，“封装得不充分，所以不是面向对象”等，世上有不少人作出类似这样的判断。道理上也许的确是这样，但面向对象无非是编程的一种工具，是不是理论上正确的面向对象不重要，是否符合 **OCP**，生产性高不高，维护性好不好，能否适应将来的更改，等等，这些才是重点。

4.3.5 **OCP与Template Method模式**

虽说使用面向对象语言的功能，可以实现 **OCP**，但也只是说有这种可能性，并不是说什么时候都能实现。当然，虽然使用了面向对象语言，却做成了一个糟糕的设计，这种情况也是屡见不鲜的。

于是设计模式登场了。正如上节所示，设计模式就是给做得好的设计起个名字，并将它们进行分类。分类中的很多设计模式之所以优秀，是因为能够经得起 **OCP** 所要求的变化。

那么，现在实际看看几种设计模式，考察一下它们都是怎样满足 **OCP** 的。

上节介绍的 **Template Method** 模式，是满足 **OCP** 的基本手段。之所以这么说，是因为其他的设计模式都是利用多个类的关联来实现的，而 **Template Method** 模式则仅仅使用了继承，基本上无非就是实现一个抽象类而已。

上节利用 Ruby 标准模块的 **Enumerable** 和 **Comparable** 解释了 **Template Method** 模式，这次从既存类中提取抽象类（Ruby 中是模块）的观点来考察一下。

表 4-4 列举了 Ruby 标准类 **IO** 的方法中用于输出的方法。

表4-4 IO 类与输出相关的方法

方法名	功 能
io<<obj	输出对象
print	输出参数
printf	带格式输出
putc	输出一个字符
puts	输出一行（换行）
write	输出字符串

这些方法在与 **IO** 类有互换性的 **StringIO** 类及 **ARGF** 对象中也有实现。但是，仔细想一想，不管哪种方法都进行类似的处理，所以可以用 **Template Method** 模式归纳一下。

例如，选择 **write** 作为基本处理，其他方法可以用表 4-5 所示的处理来实现。将这些处理作为一个模块分割开以后（参见 图 4-26），将命令行参数指定的多个文件结合成一个虚拟文件的 **ARGF**，在对字符串进行输入输出的 **StringIO** 类中，就没有必要再重复这些定义。

表4-5 用write 方法将表4-4中的方法归纳起来

方法名	使用write的处理
print	将各参数变成字符串传递给write
printf	用第一个参数将参数格式化以后传递给write
putc	将代表字符编码的整数变成字符串传递给write
puts	把参数字符串和换行符传递给write

```
module Writable
  def <<(obj)
    self.write(obj.to_s)
  end
  def print(*args)
    args.each do |obj|
```

```
        self.write(obj.to_s)
    end
end
def printf(fmt, *args)
    self.write(sprintf(fmt, *args))
end
def putc(c)
    self.write(sprintf("%c",c))
end
def puts(s)
    self.write("%s\n",s)
end
end
```

图 4-26 Ruby 中 Writable 模块的定义

也就是说，有了 **Writable** 模块，**IO** 类以外的类中，只需定义适合各自实现方式的 **write** 方法，然后将 **Writable** 模块包含进来就可以了。

有了 **Writable** 模块，处理内容相似的方法就不需要分别定义，而且，在以后开发需要具备这种输出方法的类时，还可以再利用。

现在的 Ruby 还没有这样的 **Writable** 模块，实际做一做，觉得特别好。也许在将来的版本中会导入这一模块。

同样在处理中重复编程、代码复制，都是违反软件开发中重要的 **DRY** 原则的。从 **OCP** 的观点来看，重复也是非常恶劣的。同样的代码反复出现，如果要对代码进行某种修改，那么全部的代码都必须修改。不能算作“对于修改而言是封闭的”。

数据结构的扩展也有影响复制代码全体的危险性，所以也不能算“对于扩展而言是开放的”。**OCP** 与 **DRY**，这两个原则实际上具有相同的意义。

4.3.6 Observer 模式

Template Method 模式说到底无非是继承，现在调查一下关联到多个类的设计模式与 **OCP** 的关系吧。作为最简单的例子，还以上节讲解过的

Observer 模式为例。

Observer 模式中，有降低多个对象间依赖性的机制。请看图 4-27。那是上节最后所用的程序示例。由每秒产生一次事件的被观察者（更新通知者）和显示当前时间的观察者所构成的简单时钟程序。

```
require "observer"

# 更新通知者 (Observable)
# 每秒发送1 次更新通知的类
class Tick

  include Observable
  def tick
    loop do
      now = Time.now
      changed
      notify_observers(now.hour, now.min, now.sec)
      sleep 1.0 - Time.now.used / 1000000.0
    end
  end
end

# 观察者 (Observer)
# 依照通知，表示现在时刻的类（文字版）
class TextClock

  def update(h, m, s)
    printf "\e[8D%02d:%02d:%02d", h, m, s
    STDOUT.flush
  end
end

tick = Tick.new
tick.add_observer(TextClock.new)
tick.tick
```

图 4-27 Observer 模式的程序示例，显示字符式时钟

程序的细节上节已经讲解过了，就省略不讲了，这里重要的是末尾 3 行。更新通知者与观察者的关系仅用 **add_observer** 一行来定义，其他部分全部都是互相独立的。

所以，即使以后要求变化了，需要更改时钟的外观时，只需要生成新的观察者类来代替 **TextClock**，并用 **add_observer** 登录就行了。很简单地就能进行功能的追加和更改，可以认为这个设计模式对于更改是开放的。

另外，更新通知者与观察者之间的消息只有通过 **add_observer** 而建立的唯一关系，不必担心有别的恶劣影响。这意味着对于修改而言是封闭的。

由此可知，**Observer** 模式是满足 **OCP** 的。

不仅限于 **Observer** 模式，很多被认为优秀的设计模式，都可以基于 **OCP** 来说明它们为什么优秀。

那么，不使用 **Observer** 模式的情形会怎么样呢？编写一个不用 **Observer** 模式的程序，实现与图 4-26 相同的处理吧（参见图 4-28）。

```
class TextClock
  def start
    loop do
      now = Time.now
      printf "\e[8D%02d:%02d:%02d", now.hour, now.min, now.sec
      STDOUT.flush
      sleep 1.0 - Time.now.used / 1000000.0
    end
  end
end

TextClock.new.start
```

图 4-28 不使用设计模式的文字时钟的代码，乍一看是清晰而良好的代码

看起来比图 4-27 要简短得多。但能否经得住将来的更改呢？**TextClock** 类专门设计为用文字显示时刻。**Observer** 模式版本中更新通知者所进行的处理，也就是计秒数处理和观察者的时刻显示处理（在图 4-27 中）相互连接起来了。所以，**TextClock** 处理的一部分不能被其他应用程序所沿用。估计只能将 **TextClock** 复制，然后进行改造了。复制，也就是将进行相同处理的代码分散到几处，是违反 **DRY** 原则的。

但是，如果事先知道文字时钟的规格将来也会一成不变，图 4-28 的程序还是很称心的。因为它简短而直接，执行效率也高。

归根结底，DRY 也好，OCP 也好，都不过是原则，根据具体情况，还是要做适当的选择。如果代码没有再利用的打算，也没有将来要扩展其功能的打算的话，也就没有必要生搬硬套设计模式。使用设计模式时有必要先做判断。

4.3.7 使用Strategy模式

介绍一个新的设计模式。根据《设计模式》，Strategy（策略）模式是这样解释的：“定义算法的集合，将各算法封装，使它们能够交换。利用 Strategy 模式，算法和利用这些算法的客户程序可以分别独立进行修改而不互相影响。”

Strategy 模式，就是将容易变化的处理归纳为独立的对象，然后使它们能够互相交换。使用方法与将容易变化的处理交给子类的 Template Method 模式相类似。两个模式最大的区别在于，Strategy 模式是独立的对象，能够动态交换处理逻辑（参见图 4-29）。



图 4-29 Template 模式与 Strategy 模式

静态语言中，Strategy 对象需要一个共同的父类，实现这个父类往往要用到 Template Method 模式。

通过 Ruby 库中使用 Strategy 模式的例子来介绍 `cgi/session` 库。`cgi/session` 库是 CGI 程序中，识别某一特定用户的一连串操作（session）的库。

使用 `cgi/session` 库，CGI 里有必要保存 `session` 固有的数据，但保存方法因应用程序的不同而有所不同。一个应用程序中可能是往临时目录里放一个文件，另一个应用程序中或许是将 `session` 数据放在数据库里。想使用的数据库管理器也是千差万别。

像这样的情形，不知道会有什么要求发生，库对于变化应该有开放性。`cgi/session` 库使用 `Strategy` 模式，对应预想中的未来变化。

图 4-30 是 `cgi/session` 的使用示例。其中重要的是第 6 行（即 `'database_manager'` 那一行）。

```
require 'cgi'
require 'cgi/session'
require 'cgi/session/pstore' # 提供CGI::Session::PStore

session = CGI::Session.new(cgi,
                           'database_manager' =>
CGI::Session::PStore, # 使用PStore,
                           'session_expires' => Time.now + 30 * 60)
# 30 分钟超时
if cgi.has_key?('user_name') and cgi['user_name'] != ''
  session['user_name'] = cgi['user_name'].to_s
elsif !session['user_name']
  session['user_name'] = "guest"
end
session.close
```

图 4-30 `cgi/session` 库的使用例子

`CGI::Session` 类在生成对象时，用哈希表指定选项，可以指定 `database_manager` 为保存实际数据的类（数据库管理器）。这个例子中，指定了利用 Ruby 标准版附带的简易面向对象数据库 `PStore` 的数据库管理器 `CGI::Session::PStore`。

除此之外，Ruby 标准版还提供了将 `session` 数据保存在普通文件的 `CGI::Session::FileStore` 类，以及保存在内存的 `CGI::Session::MemoryStore` 类³ 等。Ruby 标准版中没有附带将 `session` 信息保存在关系数据库（RDBMS）的数据库管理器类，但在 RAA⁴ 中，记录有利用 MySQL、PostgreSQL 以及 dRuby 进行网络通信的类。

3 因为保存在内存中，所以在使用多个进程的 Apache 中不能使用。

4 RAA (Ruby Application Archive) 是一个索引 (<http://raa.ruby-lang.org/>)，记录了利用 Ruby 实现的库及应用程序。利用 MySQL 的，可以参照

<http://moko.cry.jp:3232/~keiji/linux/pub/files/mysql-session-0.16.tar.gz>。利用 PostgreSQL 的，可以参照 http://www.angelfire.com/vi/oremacs/ruby_page.html。利用 dRuby 的，可以参照 <ftp://ftp.tietew.jp/pub/ruby/drbsession-0.1.tar.gz>。

`cgi/session` 对象在初始化处理中生成数据库管理器类的对象，保存在类实例变量 `@dbman` 中。在需要访问 `session` 数据的时候，调用数据库管理器的方法进行实际的处理（参见表 4-6）。

表4-6 数据库管理器的方法

方法名	功 能
<code>restore</code>	读入 <code>session</code> 数据
<code>update</code>	写出 <code>session</code> 数据
<code>close</code>	关闭 <code>session</code> 数据

数据库管理器的 `restore` 方法返回表示实际数据库的对象，`CGI::Session` 对象使用与该数据库对象数组相同的接口（`[]` 方法和 `[]=` 方法）来访问。`update` 方法将数据库对象的状态写到实际文件里。Ruby 这样的动态语言，只要有了表 4-6 所示的这些方法，不管什么都能作为数据库管理器类来使用。

4.3.8 Strategy 模式与 OCP

最后来验证一下 Strategy 模式是如何满足 OCP 的。

首先看是不是开放的。使用 Strategy 模式时，只要更换封装了算法的 Strategy 对象，就可以追加及修改功能。`cgi/session` 的情形，只要更改了选项所指定的数据库管理器类，就能够改变 `session` 数据的保存方法。所以，可以说 Strategy 模式对扩展而言是开放的。

另一方面，即使扩展或修改了功能，利用 Strategy 模式的一方也没有必要改变。`cgi/session` 的情形，即使开发了新的保存 `session`

数据的方法，既有的代码也没有必要变更，而且利用新的数据库管理器时，除了 `CGI::Session` 对象的初始化选项以外，别的代码也没有更改的必要。所以，`Strategy` 模式对于更改而言是封闭的。

因此，`Strategy` 模式完全满足 `OCP`。

由于篇幅的关系，设计模式中只验证了很小一部分，世上很多设计模式，为了能应对将来可能的修改，都是按照 `OCP` 的要求来设计的。

* * *

本节从 `OCP` 的观点，重新审视了一下设计模式。由继承实现的 `OCP`，通过使用设计模式而变得更强大有力。我想大家可以认识到设计模式是应变能力很强的工具，得到了广泛应用。

实际上，`OCP` 与设计模式相结合，并不是我的首创。据我所知，日本国内将这两者结合起来讲解的，还有石井胜先生。石井先生的讲解详见

<http://www.objectclub.jp/community/memorial/homepage3.nifty.com/masarl/article/dp-ocp-2.html>。

词汇与通用语言的重要性

关于设计模式的书籍刚出现的时候，我最初的印象是“夸夸其谈，其实都是些理所当然的内容”。《设计模式》一书所列举的 23 种模式，很多都是经常使用，并不怎么罕见的模式。而且，还含有很多在 `C++` 及 `Java` 那样的静态语言中虽然有效，但在 `Smalltalk` 及 `Ruby` 中却并没有多大意义的模式。

但过了一阵子我有了更深刻的认识。设计模式的本质，并不是介绍至今没有用过的新模式，而是通过给屡屡使用过的模式起一个合适的名字，从而提供了设计时的词汇。

人使用语言进行思考。没有语言的系统也就无法思考，不使用语言，也不能与其他人进行交流。人类的生存与人际关系都是跟语言密切相关的。

旧约全书里记载了巴比伦塔的故事，描写了人与人之间因为语言互不相通，而中止了塔的建设，人们都纷纷散去的场面。语言的重要性可见一斑。

据说住在阿拉斯加与加拿大的因纽特人的语言中，表现雪与冰的词汇有 80 种。居然还需要把这么微妙的差异加以区别。我们也有很多表现雨的词汇，如毛毛雨、初夏雨、秋雨、阵雨等。

设计模式也一样。软件开发中，虽然也存在能用于各种局面的类或专业用语，但如果不给它们起一个合适的名字，很多开发人员都不可能意识到它们的存在。设计模式就是将这样的词汇集中起来，并进行分类的一种尝试。所以，书上介绍的 23 种模式，不过是一个开始。最终还应有一个能收录更多模式的“设计模式目录”。

很遗憾，这种目录还没有出现。也许人们连现有的 23 种模式都没能够灵活运用，还没进化到需要追求更多模式的阶段。

第5章 Ajax

5.1 Ajax 和 JavaScript（前篇）

很多网络应用程序，比如 Google Maps，不用进行网页更新就可以将处理进展下去。

如果要进行网页更新，就要对画面进行描绘处理，应用程序的反应会变得迟缓。另一方面，使用 Web 2.0¹ 技术的应用程序，因为不需要进行画面更新，能顺利地将画面处理下去。像上述这一类实现 Web 应用的技术，称为 Ajax（Asynchronous JavaScript and XML），含义是“异步 JavaScript 及 XML”。本章讨论 Ajax 及其相关技术。

¹ Web 2.0 不存在确切的定义，一般是指比 Web 技术更加重视扩大对使用者提供的服务。

实际上，Ajax 不算一个新技术，只是既存技术的组合。Ajax 这个名字是在 2005 年 2 月由美国 Adaptive Path 公司的 Jesse James Garrett 命名

的。现在的 Web 应用大多数都使用了 Ajax 技术，与这个名字有很大关系。名字是非常重要的，顺便提一句，美国有一种叫 AJAX 的厨房洗涤剂，欧洲有一家名为 AJAX 的足球俱乐部。

5.1.1 通信及异步页面更新

Ajax 的最大特点是进行异步操作。异步意味着，Web 浏览器的通信和页面更新是互相独立的。

以前的 Web 应用程序，每按下一个按钮就开始显示下一个页面，在页面完整呈现之前，用户只能等待，无法进行其他操作。

相反，使用了 Ajax 技术的页面是在后台和 HTTP 服务器进行通信。设计优良的 Web 应用程序，在客户和服务器通信的过程中也可以让用户进行操作，而不需要等待。Ajax 的最大优点就是改善了应用程序的操控性。

以前的 Web 应用程序和使用了 Ajax 的程序的区别，我们用图来说明。对于以前的应用程序，Web 浏览器和 HTTP 服务器是按如图 5-1 所示的方式进行通信的。

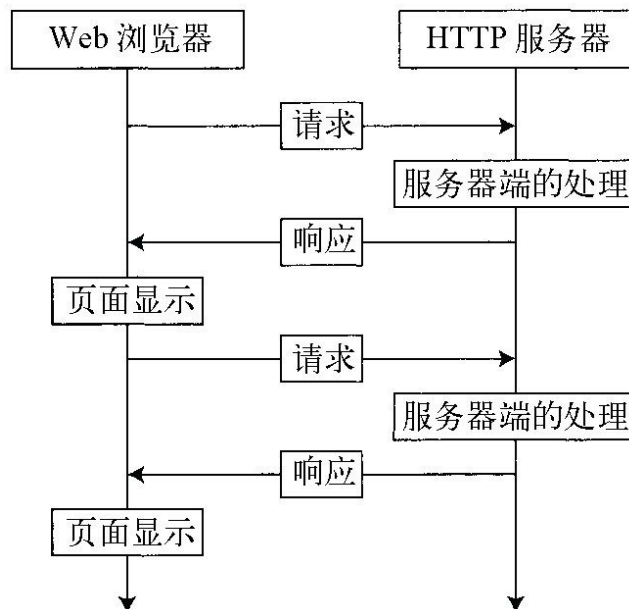


图 5-1 以前的 Web 应用程序的操作

每当用户进行操作之后，需要更新页面，在得到服务器的响应之后，下一个页面才能显示出来。在这之前，用户只能等待。

Ajax 应用程序则是按图 5-2 所示的方式进行操作的。

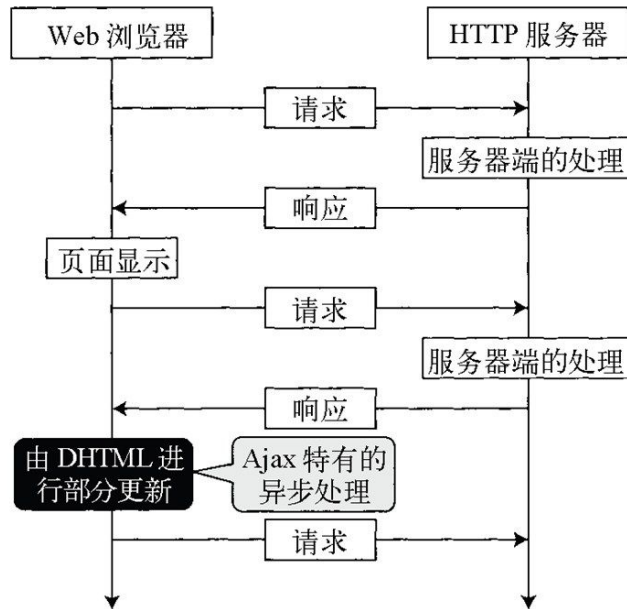


图 5-2 Ajax 应用程序的操作

Ajax 技术中，对于用户进行的操作，基本是由 JavaScript 在 Web 浏览器中进行响应。仅在数据必须从服务器获取的情况下，才在后台进行异步通信。在通信过程中，用户也可以继续对 Web 浏览器进行操作。和服务器通信得到的结果由 DHTML（后面我们将进行介绍）对当前的页面进行部分更新而显示出来。

比如 Google Maps，在对地图进行拖拉操作时，感觉像是在操作一个本地机上的巨大的地图文件，滚动得非常顺利（参见图 5-3）。实际上，这是使用了 Ajax 技术从服务器异步获取分割得很小的地图图像。如果在速度较慢的网络上使用 Google Maps，在通信结束之前，还没来得及获取地图图像数据时，地图的一部分会显示空白。



图 5-3 Google Maps 的画面

异步通信技术不仅仅是使用在 Web 应用程序上，对于改善应用程序的反应也非常有效。异步通信是将处理细分成多个处理来交替执行。处理时间合计起来可能会更长。但是，在通信结束之前用户不需要等待，所以操作的不愉快感会少很多。一般情况下，相对于减少总处理时间来说，减少最大等待时间更重要。

5.1.2 技术要素之一：JavaScript

接下来，我们依次讨论支撑 Ajax 的 3 个主要技术：JavaScript、XML 及 DHTML。

JavaScript 是最近几乎所有的 Web 浏览器处理系统都支持的一种编程语言。因此，有时候它也被称为“世界上最普及的编程语言”。

JavaScript 可以简单地嵌入到表示网页的 HTML 中去。图 5-4 显示了一个嵌入到 HTML 中的 JavaScript 的简单例子。如果将图 5-4 中的 HTML 文件读入到 Web 浏览器中，会显示出“请点击”的字符串。用鼠标点击此字符串会弹出一个信息窗口，显示“已经点击了哦”。

```
<html><head><title>JavaScript 举例 </title>
<script type="text/javascript">
<!--
function alertOnClick() {
```

```
    alert("已经点击了哦");
}
// -->
</script></head>
<body>
<a onclick="alertOnClick()"> 请点击 </a>
</body>
</html>
```

图 5-4 嵌入到 HTML 中的 JavaScript 示例

像上述这样，利用 JavaScript 可以做出完全不用和服务端进行通信的网页。如果将 JavaScript 技术发挥到极致的话，仅仅使用 Web 浏览器就能实现令人吃惊的操作。

JavaScript 本来是一种用来控制 Web 浏览器的脚本语言，是由美国 Netscape 通信公司设计的，设计者是 Brendan Eich。一开始它叫做 LiveScript，因为当时 Java 技术变得越来越流行，于是改名为 JavaScript²。

2 JavaScript 这个名字不是随便起的，是和美国 Sun 公司进行交涉，解决了商标上的问题之后才得以使用的。

实际上，JavaScript 和 Java 是完全不同的语言。除了名字和语法有相似之处，语句都是由大括弧{ }括起来之外，再没有其他关联性。面向对象编程方面也完全不同。

另外，JavaScript 也被称为“最易被误解的编程语言”³。比如它曾被认为是嵌入式应用程序语言中简化的编程语言之一。因为它的语法规则比较简单，所以更易被误解。

3 参考 JSON 的开发者 Douglas Crockford 的文章 *JavaScript: the World's Most Misunderstood Programming Language* (<http://javascript.crockford.com/javascript.html> , <http://d.hatena.ne.jp/brazil/20050829/1125321936>)。

实际上，像我这样的“编程语言迷”，对 JavaScript 有更大的兴趣，使用起来也格外顺手。对于 JavaScript 的详细讨论会放在后面的章节继续进行。

5.1.3 技术要素之二：XML

也许没有必要对 XML（eXtensible Markup Language）重新进行说明。它是和 SGML、HTML 类似的，使用标签（tag）对数据进行标识说明的一种语言。

现在，XML 已经成为了在数据表示、配置各种文件及其他多种场合下广泛使用的一种格式。

Ajax 的名字中部分包含了 XML，是因为当初大部分使用 Ajax 技术的应用程序都使用了 XML 数据，还有，用 JavaScript 进行异步通信的对象的名字是 XMLHttpRequest。

可是，不使用 XML 的 XMLHttpRequest 的通信也是存在的。使用 Ajax 技术的 Web 应用进行通信的数据格式也是多种多样的，比如有普通文本格式和下面将要介绍的 YAML，以及 JSON 等。

5.1.4 XML 以外的数据表现形式

最近，YAML 和 JSON 作为数据表现形式受到了人们的关注。我们通过与 XML 作对比来进行说明。

从 YAML 的全称（YAML Ain't Markup Language）可以看出它不是标识语言⁴。XML 是可以作为数据表现形式的标识语言，而 YAML 只是表示数据的语言。YAML 的目的仅仅就是表示数据，和 XML 相比有以下特点：

4 软件界经常使用在缩写中包含自身的回归名称，有代表性的例子有 GNU（GNU's Not Unix）和 PHP（PHP:Hypertext Processor）。

- 记述简洁；
- 容易理解；
- 专注于表示数据，不用费心考虑给标签起名字。

通过实际的例子我们就能感受到上述特征。图 5-5 是有 name、job、kids 3 个项目的表（Ruby 中称哈希表），kids 的内容是 4 个元素的列

表（Ruby 称数组）。用 Ruby 的数据结构表示如图 5-6 所示。

```
name: Matsumoto
job:  Programmer
kids:
  - Girl
  - Girl
  - Boy
  - Girl
```

图 5-5 YAML 数据示例

```
{
  "name"=>"Matsumoto",
  "kids"=>["Girl", "Girl", "Boy", "Girl"],
  "job"=>"Programmer"
}
```

图 5-6 YAML 数据的 Ruby 表示

Ruby 配置了标准的 YAML 对应库，可以直接读写 YAML 数据（参见图 5-7）。

```
yaml = "... " # 图5-5 的YAML 数据字符串
p YAML.load(yaml)
```

图 5-7 用 Ruby 读取 YAML 数据的代码

和 YAML 相提并论的另一种数据表现形式是 JSON（JavaScript Object Notation），意思是 JavaScript 对象表示法。发音为“J-song”。顾名思义，JSON 是直接把 JavaScript 表示对象的程序拿来记述数据了。JSON 是合法的 JavaScript 程序，作为 JavaScript 实现可以生成对象⁵。

5 从没有信赖的服务器传来的 JSON 数据，在执行之前需要对数据内容进行检查。否则可能引起安全问题。

图 5-5 中的 YAML 数据用 JSON 表示的话，如图 5-8 所示。

```
{  
  "name": "Matsumoto",  
  "kids": ["Girl", "Girl", "Boy", "Girl"],  
  "job": "Programmer"  
}
```

图 5-8 JSON 数据示例

JSON 采用的对象表示法基本和 Python 语法相同，和 YAML 也有很多共通之处。YAML 允许将表的项目名用引号括起来。如果把 JSON 的数据传给 YAML 解析器，一般都能正确解释。

JSON 可以表现下面 6 种数据类型：数值（整数及浮点小数）、字符串、布尔值（真、假）、数组、对象（键和值的表）和 null。

YAML 通过扩展可以表示各种形式的数据。相比较，JSON 就显得太简单了。但实际上，有这 6 种数据类型应该就足够了。

5.1.5 技术要素之三：DHTML

DHTML，动态 HTML，顾名思义，可以动态地对 HTML 进行引用、修改和更新。更具体地说，它是利用装载在网页中的 JavaScript，使用 DOM（文档对象模型）对网页数据进行操作。使用 DOM 可以进行下述处理：

- 取得页面中特定标签中的数据；
- 修改标签的数据（文字、属性等）；
- 在页面中添加标签；
- 设定事件处理程序。

5.1.6 JavaScript 技术基础

在本章后半部分会针对 Ajax 的核心技术 JavaScript 进行详细讲解。

JavaScript 是以对象为基础的语言。也就是说，所有的数据都可作为“对象”进行统一处理。不过，它不具备“类”这样的所谓普通面向对象语言所提供的功能。后面还要说明，即使去除 JavaScript 面向对象的编程功能，它也可以作为普通的结构化编程语言来使用⁶。

6 JavaScript 的专用语中包含了“class”。这也许是预留，以便将来可能导入以类为基础的面向对象编程。

只要具备某种编程语言的经验，就会很容易掌握 JavaScript 的基础。下面就以已具备某种语言经验的人为对象，对 JavaScript 最基础的部分进行讲解。

名字

区分大小写字母。变量名可以由英文、数字、下划线（_）及美元符号（\$）组成。

注释

由//开始的行是注释语句。也可以使用 C 语言中以/* 开始和以*/ 结尾的注释方式。

保留字

表 5-1 列出了 JavaScript 的保留字，共 53 个，对于“小语言”来说真不少。这些保留字不能用作函数名或变量名。其中一些词目前并没有使用，是为将来可能的使用而预留的，比如用于类型指定的一些词：byte、int、float 等。

表5-1 JavaScript专用语一览

abstract	continue	float	int	public	throws
boolean	default	for	interface	return	transient
break	do	function	long	short	true
byte	double	goto	native	static	try
case	else	if	new	super	var
catch	extends	implements	null	switch	void
char	false	import	package	synchronized	while

class	final	in	private	this	with
const	finally	instanceof	protected	throw	

基本语法

JavaScript 的基本语法和 C、Java 类似。最大的不同是，JavaScript 不指定变量类型。

运算符

和 C 语言类似，优先顺序也基本相同。

函数定义

JavaScript 的特点之一是把函数作为对象进行处理。C 也是将函数作为对象处理，但 JavaScript 的不同之处在于函数对象有闭包（closure）⁷，可以使用函数外面的局部变量。这个闭包功能成为了 JavaScript 面向对象功能的基础。

7 闭包是 Ruby 中的块变为对象后的结果。它的优点是，只要闭包还存在，就能访问闭包内的变量。

生成函数对象用 **function** 语句，如图 5-9 所示。

```
function 函数名（参数）{  
    函数定义  
}  
  
var 变量名=function（参数）{  
    函数定义  
}
```

图 5-9 function 语句的使用方法

在两个例子中，后者生成匿名函数对象，然后赋值给变量；前者生成的函数有函数名，内部功能是一样的。

属性

JavaScript 可以给任意的对象粘连有名字的对象，这称为对象的属性。访问属性时使用“.”，如图 5-10 所示。

```
o = new Object(); // 生成对象
o.x = o.y = 15;   // 设定对象属性
o.sum = o.x + o.y; // 访问属性
```

图 5-10 属性的使用方法

5.1.7 原型模式的面向对象编程语言

JavaScript 是面向对象编程语言。当初我认为它只是容易嵌入到应用中的一种简单语言，但实际上它具备了完善的面向对象功能。当我听说它具有原型模式的面向对象功能时，吃惊地差点从椅子上摔下来。

如果具有原型模式的面向对象功能的话，就可以最大限度地消减语言本身的固有功能。这非常适合于 JavaScript 这样的语言。JavaScript 的设计者具有相当好的直觉。

以类为中心的传统面向对象编程，是以类为基础生成新对象。类和对象的关系可以类比成铸模和铸件的关系。

而原型模式的面向对象编程语言没有类这样一个概念⁸。

⁸ 使用后面讲到的 `prototype.js` 库，可以生成名为 `Class` 的功能和类相同的对象。

需要生成新的对象时，只要给对象追加属性。设置函数对象作为属性的话，就成为方法。当访问对象中不存在的属性时，JavaScript 会去搜索该对象 `prototype` 属性所指向的对象。

JavaScript 利用这个功能，使用“委派”⁹ 而非“继承”来实现面向对象编程。

⁹ 委派是指，把对于某个对象的调用传送到另一个对象上。

图 5-11 列举了一个用 JavaScript 实现的狗的例子¹⁰。在第 4 章我们介绍过用原型模式的 Io 语言实现了相同的程序。与它相比，图 5-11 中

的程序感觉上介于类模式和原型模式之间。虽然这个例子没有什么实用性，但是可以从中感受到 JavaScript 的面向对象编程的氛围。

10 读者可能把对用狗或其他哺乳动物作例子已经厌烦了。这一点我也有同感，但是也举不出更好的例子。

```
// 生成Dog。... (A)
function Dog(){
    this.sit = function () {return "I'm sitting"}
}
// 从Dog 生成对象dog... (B)
var dog = new Dog()
// dog 是狗，所以能 sit... (C)
alert(dog.sit())
// 生成新型myDog... (D)
function MyDog () {}
// 指定委派原型
MyDog.prototype = new Dog()
// 从MyDoc 生成新对象myDog... (E)
var myDog = new MyDog()
document.write(myDog.sit())
```

图 5-11 原型模式编程的示例

我们从程序的一开始仔细来看。（A）定义了狗对象的原型函数 **Dog**。让人吃惊的是 JavaScript 使用了函数对象来代替类。函数对象起到了对象构造器的作用¹¹。执行构造器的处理时，给 **this** 追加新的属性，从而完成对象的初始化。

11 构造器是指给对象进行初始化的函数。

（B）为了从原型生成对象，使用了 **new** 语句。和 Java、C++ 是类似的。但是 **new** 的处理内容有很大的不同。

原型 **Dog** 调用 **new** 完成了以下处理：

1. 生成对象；
2. 将委派原型的内部属性（**__proto__**）设置为 **Dog.prototype**；

3. 调用函数 **Dog**，参数即为传递给 **new** 时的参数；

4. 返回新生成的对象。

(C) 调出方法。取出对象 **dog** 的 **sit** 属性，执行这一属性的函数对象。

函数作为对象的属性被取出和调用时，拥有该属性的对象会被自动设置成 **this**，在函数中可以用 **this** 访问该对象。

那么，继承功能是如何实现的呢？JavaScript 中的继承风格完全变了。

首先，像 (D) 那样定义原型函数。目的只是定义一个子类，所以定义是空白的。子类自身的初始化由 **MyDog** 函数完成。

然后，把想要继承的类的对象赋值给新原型对象的 **prototype** 属性。请注意，要赋的值不是定义父类功能的原型函数，而必须是由父类原型函数生成的对象。这就是所说的原型模式。

这样，对由新定义原型生成的对象来说，当访问它不知道的属性时，就会委派给对象 **dog**。JavaScript 通过这种方式实现了类模式语言中相当于继承的功能。

实现继承之后，其他是类似的。(E) 利用 **new** 运算符生成新对象，并调用方法。

这样，JavaScript 用较简单的方式可以实现各种功能，这让人不禁想到 **Lisp**。不过，不能否认的是，JavaScript 的方式过于简单反而使记述太过繁杂，它不能像 **Lisp** 那样用宏定义隐藏复杂性。

5.1.8 使用prototype.js库

为了克服 JavaScript 记述过于繁杂的缺点，JavaScript 提供了进行功能扩展的一些库。在这里介绍 **prototype.js** (Prototype JavaScript Framework)。

prototype.js 是由 Sam Stephenson 开发的，是可以在 MIT 许可证下使用的库¹²。

12 从 <http://prototype.conio.net/> 中可以得到。

prototype.js 受到了 Ruby 的影响，方法名和功能都和 Ruby 有些相似。Ruby 用户可能会觉得很熟悉。实际上，Ruby on Rails 中标准地附加了 prototype.js 库，使用得很广泛。而且在 script.aculo.us (<http://script.aculo.us/>) 和 Rico (<http://openrico.org/>) 中也被作为 JavaScript 的函数库基础来使用。

5.1.9 prototype.js的功能

这里，我们简单介绍一下 prototype.js 的各种功能。

Ajax功能

prototype.js 支持 XMLHttpRequest 对象，可以对 HTML 进行异步更新。具体来说，像下述的语句就可以获取 XMLHttpRequest 对象。

```
new Ajax.Request(url,options)
```

实际上，不同的 Web 浏览器获取 XMLHttpRequest 对象的方法也是不同的，比如 IE6.0 之前与之后的版本，或者同样是 IE，因为使用的 MSXML ActiveX 对象的版本不同等，都需要对它们区别对待。Ajax.Request 帮我们屏蔽了这些与代码移植相关的问题。

作为 prototype.js 的 Ajax 功能，其他还有异步更新 Ajax.Updater 和定期异步更新 Ajax.PeriodicalUpdater。

Enumerable

这个名字看起来很眼熟吧？它是 Ruby 的 Enumerable 模块在 JavaScript 中的实现。包括了 each、collect、select、detect 以及 inject 等常见的方法名。prototype.js 对原有的 Array 类追加了 Enumerable，并扩展了它的功能，所以对 Array 也可以调用这些方法。

使用 prototype.js 的 Enumerable 时需要注意以下几点：

- JavaScript 没有块¹³的概念，可以把函数对象作为参数来代替块；

13 块在 Ruby 中可以追加到方法调用的末尾，作为代码块来处理。附加块的方法在被调用时可以调用块中定义的内容。

- JavaScript 中由复数单词组成的变量名，不是用下划线进行连接，而是将单词的首字母大写。例如，使用 `findAll`、`sortBy`，而不是 `find_all`、`sort_by`；
- 方法名中不能使用“!”和“?”。

`Enumerable` 提供的方法如表 5-2 所示，与同名的 Ruby 中的 `Enumerable` 方法功能相同。

表5-2 `Enumerable` 方法一览

<code>all([iter])</code>	<code>grep(pattern[, iter])</code>	<code>pluck()</code>
<code>any([iter])</code>	<code>include(obj)</code>	<code>reject(iter)</code>
<code>collect(iter)</code>	<code>inject(init, iter)</code>	<code>select(iter)</code>
<code>detect(iter)</code>	<code>invoke()</code>	<code>sortBy(iter)</code>
<code>each(iter)</code>	<code>max([iter])</code>	<code>toArray()</code>
<code>entries()</code>	<code>member(obj)</code>	<code>zip(ary...)</code>
<code>find(iter)</code>	<code>min([iter])</code>	
<code>findAll(iter)</code>	<code>partition([iter])</code>	

但是，它提供了 Ruby 中不存在的 `pluck` 和 `invoke` 方法。它们的功能如下：

`pluck` 集合了各要素指定的属性（参见图 5-12）。

```
def ary.pluck(property)
  ary.map{|x| x[property]}
end
```

图 5-12 Ruby 中对于 `pluck` 的定义

invoke 集合了各要素调用方法时得到的结果（参见图 5-13）。

```
def ary.invoke(method, *args)
  ary.map{|x| x.send(method, *args)}
end
```

图 5-13 Ruby 中对于 **invoke** 的定义

Enumerable 方法的用例如图 5-14 所示。

```
var ary = [1,2,3,4,5]
var ary2 = ary.collect(function(x){
  return x*2
})
ary2.each(function (x) {
  document.write("item: "+x+"<br>")
})
```

图 5-14 **Enumerable** 方法的示例

使用 **Enumerable** 的功能写出的程序会让 Ruby 的用户觉得容易理解。不过，有时候也会觉得受到了语法的限制，比如：“**function** 这个保留字太长了”，“小括号和大括号怎么混在一块儿了呢”。

其他扩展功能

prototype.js 的扩展功能涉及到其他各种各样的领域，限于篇幅不可能在这里全部进行介绍。在此仅说明其中一些重要功能。

首先，使用 **Object.extend()** 可以给对象追加功能。例如，下面是给自己生成的对象 **obj** 追加 **Enumerable** 功能：

```
Object.extend(obj, Enumerable)
```

或者，像下面这样给既有的类（函数对象）追加 **Enumerable** 功能：

```
Object.extend(Array.prototype, Enumerable)
```

Function 类的功能也得到了扩展。**JavaScript** 的函数对象作为个体来说，没有保存处理对象的信息。可是当作为回调函数使用的时候，有时候希望函数对象能够保持处理对象的状态。为了实现这一点，**prototype.js** 给 **Function** 类追加了 **bind** 方法。

前面说过 **Array** 类里追加了 **Enumerable** 功能，除此之外，**Array** 类里还追加了其他一些 **Ruby** 常用的方法。

另外，**String** 类、**Number** 类以及构成 **DOM** 用的一些类群里也追加了方法。最后还提供了一些函数的缩写形式。

缩写形式的函数有，从 **id** 取得 **DOM** 元素用 **\$()**、取得 **Form** 值用 **\$F()**、变换成 **Array** 用 **\$A()**、生成哈希值用 **\$H()**、生成范围对象用 **\$R()** 等。

* * *

以上讲解了 **Ajax** 的概要和组成它的核心技术。**Ajax** 不是一个新技术，但是它很好地组合了既有的 **JavaScript** 和 **DHTML** 技术，在 **Web** 浏览器上实现了丰富的客户端应用。

Ajax 已经是人尽皆知了，成为了 **Web 2.0** 必不可少的技术。

5.2 Ajax 和 JavaScript（后篇）

有很多用 **Ajax** 制作的网站，使用起来都很方便。它们都是用 **Web 2.0** 技术实现的。典型的例子就是 **Google Maps**。**Google Maps** 的特点是，对于指定的地图，不仅是显示一页画而已，对地图进行拖拉操作时滚动得很平滑，就好像是在阅览一张已经下载了的大地图一样方便。

对地图进行放大、缩小也很方便，好像不是用 **Web** 浏览器而是用专门的软件来操作地图那样好用。在 **Google Maps** 之前的地图网站没有这

样方便的，每当移动地图、改变比例尺大小时，都要在点击之后等待重新显示画面。

除了地图之外，还有简单的工作计划管理等也采用 Ajax 技术的网站，此类站点有很多很多。例如，开发了 Ruby on Rails 的 37signals 做的 ToDo 管理网站“Tada List”（参见图 5-15）。



图 5-15 可以管理工作计划的网站 Tada List

在这个网站上，添加了工作计划（ToDo）后，就会显示一个动画，还可以通过拖拉操作给项目排序。

像这样的 Ajax 网站，它们有 3 个共同特点：

1. 没有 Web 页面跳转；
2. 通过异步通信实现快速反应；
3. 实现了动画和拖拽等单独使用 HTML 格式无法表现的用户界面。

构成 Ajax 的基本技术早在 1997 年就已经确立下来了。到 2006 年左右，计算机性能得到大幅度的提高，这一技术开始引人注目。构成 DHTML 基础的 JavaScript，真是一个速度不怎么快的语言。当时的计算机大概难以实现足够的反应速度。

得益于计算机的性能提高和软件开发技术的进步，某种技术在开发出来之后经过相当长一段时间才得到普及，诸如此类的例子并不少见。

20 世纪 70 年代在美国施乐公司的帕洛阿尔托研究中心（PARC）诞生的 GUI 技术，花费了近 20 年的时间才得到广泛使用。

和我有渊源的编程语言 Ruby，当初仅仅用于生成个人用的工具，因为这样在执行性能上不会有问题。但是最近几年，大规模企业系统的开发也开始采用它。这反映了随着计算机性能的提高，相对于执行性能而言，开发效率更受到重视。

在我身边，不断听到有人想在业务领域更多地使用 Ruby，让我越来越深切地感受到这种变化趋势。

5.2.1 巧妙使用DHTML

Ajax 技术的核心要素，前面已经介绍过有 JavaScript、XML 和 DHTML。下面，更进一步地看几个例子。

DHTML 亦被称为 Ajax 的本质技术。DHTML，顾名思义就是可以动态地访问、更新 HTML。具体地说，就是利用嵌入到网页中的 JavaScript，使用 DOM¹ 操作页面数据。

1 DOM 是操作 HTML 和 XML 的规范。特点是把 HTML 和 XML 作为树结构进行处理。由 W3C（World Wide Web Consortium）定义的 DOM 规范称为 W3C DOM。

DHTML 是在 JavaScript 中使用 W3C DOM，将 HTML 作为一种树结构进行操作。关于树结构，我们稍后再详细说明。更进一步讲，通过对用户输入的鼠标点击等事件指定相应的 JavaScript 函数，可以对动作（action）进行定义。

通过 W3C DOM 让 JavaScript 操作 HTML 的例子见图 5-16。将图 5-16 的 HTML 文件用 Web 浏览器打开，会显示出“请点击”的字符串。点击字符串之后，<div id="result"></div> 内的文字发生改变。

```
<html><head><title>JavaScript 举例</title>
<script type="text/javascript">
<!--
function displayOnClick() {
    var result = document.getElementById("result").childNodes[0];
    result.nodeValue = "已经点击了";
}
```

```
// -->
</script></head>
<body>
<a onclick="displayOnClick()">请点击</a><br>
<div id="result">显示会变化</div>
</body>
</html>
```

图 5-16 嵌入了 JavaScript 的 HTML 示例

图 5-16 的后半部分，对标签 **a** 的 **onclick** 属性进行指定。标签范围内的文字被点击时执行指定的 JavaScript 的 **displayOnClick()** 函数。

函数 **displayOnClick()** 使用 **id** 取出页面中相当于 **div** 部分的节点（**getElementById**），通过改变子节点的值来更新字符串。节点的概念将在后面和树结构一起说明。

前半部分 **script** 标签里用 **<!--** 和 **-->** 围起来的 HTML 注释是 JavaScript 的代码²。

2 注释部分最后一行的开头加上 **//**，是为了避免浏览器把 **-->** 也解释成 JavaScript。这种做法是为了让不支持 JavaScript 的 Web 浏览器不至于直接把 JavaScript 代码显示出来。不过，这里介绍的 HTML 文件都必须使用那些支持 JavaScript 的 Web 浏览器。在后面的例子中我们就不再加 **//** 了。

像这样通过使用 DHTML，服务器端不需要准备 CGI 就可以实现动态网页。

图 5-16 的 HTML 被读取之后，内部会生成如图 5-17 所示的树结构。树结构中，**document** 相当于树根，**div** 称为节点。

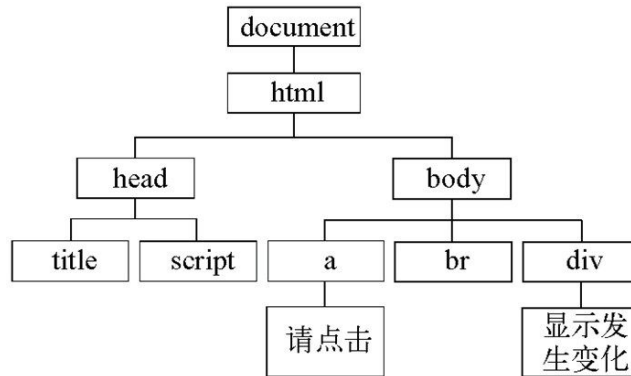


图 5-17 显示图 5-16 数据的树结构

使用 JavaScript 对树结构进行操作就是 DHTML 的本质。因此 JavaScript 提供的 W3C DOM API 有如下的功能：

- 获取 document 节点；
- 获取和更新标签数据（包括文字、类型以及属性等）；
- 追加 document 节点；
- 设定事件处理程序（event handler）³。

³ 事件处理程序是指事件发生时执行的程序。

5.2.2 获取document节点

利用 DOM 的 API 对 HTML 文件进行操作，如图 5-17 所示，首先从 DOM 树中取出节点。可以利用下面两个函数取出节点。

getElementById(name)

getElementById() 将得到标签 id 为 name 的节点对象。

getElementsByTagName(name)

getElementsByTagName() 将得到标签名字为 name 的标签节点。

一般的文件中会存在多个同样的节点，所以

getElementsByTagName() 将返回节点对象数组。当取得第 n 个项目时，在数组后附加[n]。

5.2.3 获取和更新标签数据

获取节点对象之后，通过调用对象的方法，读写对象的属性等就能够获取和更新标签数据。这些数据包括标签信息、类型、文字和事件处理程序等。

例如，标签的类别由 **tagName** 属性来得到：

```
node.tagName
```

也可以变更决定标签显示方式的 **style** 属性。设定时先读出节点的 **style** 属性，然后设定 **style** 对象的属性值。比如，想让 **node** 节点看不见的话，可以进行以下的设定：

```
node.style.visibility = 'hidden'
```

style 对象使用了和 CSS 的 **style** 属性相同的名字。不过两者属性名字的写法有一定的变化。具体说，就是去掉“-”，将单词的首字母换成大写。例如，CSS 属性中的 **border-style**，在 JavaScript 中该属性名称变为 **borderStyle**。

改变标签中的文字内容，可以从标签中取出对应于文字部分的节点（文字节点），然后指定该节点的 **nodeValue** 属性。图 5-16 的 HTML 就使用了这个方法，取出的 **div** 标签的最初节点就是文字节点。

5.2.4 设定事件处理程序

指定对应于某一事件的事件处理程序有两个方法，其中之一是如图 5-16 那样指定标签的属性。图 5-16 是把 JavaScript 函数名设置为 **a** 标签的 **onclick** 属性值来指定事件处理程序。

另一个方法是把函数设置为 JavaScript 对象的属性。具体如下面的形式：


```
对象.事件处理程序名 = function() {  
    .....  
}
```

在程序中指定事件处理程序或者替换事件处理程序的时候，用第 2 个方法更方便。图 5-18 是对图 5-16 进行的修改，在事件处理程序中再次替换了事件处理程序。

```
<html><head><title>JavaScript 举例</title>  
<script type="text/javascript">  
function displayOnClick() {  
    var result = document.getElementById("result").childNodes[0];  
    result.nodeValue = "已点击";  
    var atag = document.getElementById("alink");  
    atag.onclick = function() {  
        result.nodeValue = "已返回";  
        atag.onclick = displayOnClick;  
    }  
}  
</script></head>  
<body>  
<a id="alink" onclick="displayOnClick()"> 请点击 </a><br>  
<div id="result"> 显示会改变 </div>  
</body>  
</html>
```

图 5-18 通过属性设定事件处理程序

因为改写了点击这个事件处理程序，和图 5-16 不同的是，每次点击都会改变显示的文字。

JavaScript 可以访问函数外面的变量，内部的事件处理程序函数可以省略获取 result、atag 等节点对象的处理。

W3C DOM 规定的事件处理程序名如表 5-3 所示。

表5-3 事件处理程序一览

事件处理程序名	发生时机
onabort	加载中断时

onblur	焦点丢失时
onchange	表单内容改变时
onclick	点击时
ondblclick	双击时
onerror	出错时
onfocus	获取焦点时
onkeydown	键被按下时
onkeypress	键被按下时
onkeyup	离开按键时
onload	加载时
onmousedown	鼠标按钮按下时
onmousemove	鼠标光标移动时
onmouseout	鼠标光标离开时
onmouseover	鼠标光标重合时
onmouseup	鼠标按钮松开时
onreset	表单重置时
onresize	窗口大小改变时
onscroll	滚动时
onselect	选择时
onsubmit	表单提交时
onunload	页面终止时

5.2.5 追加标签节点

至此，我们说明了如何变更树结构中既有的属性，W3C DOM 还可以对树结构本身进行操作。

用 `appendChild` 方法可以为节点对象追加节点，消除节点用 `removeChild` 方法。图 5-19 的 HTML 文件，每次按键点击都会增加一个文字节点。点击按键之后，在末尾就追加了新的文字。

```
<html><head><title>JavaScript 范例</title>
<script type="text/javascript">
function appendText() {
    var body = document.getElementsByTagName("body")[0]
```

```

    var p = document.createElement("p")
    p.appendChild(document.createTextNode("已点击。"))
    body.appendChild(p)
}
</script></head>
<body>
<input type="button" value="点击！" onclick="appendText()"/>
</body>
</html>

```

图 5-19 可以追加文字节点的 HTML

5.2.6 本地HTML应用

好了，采用以上讲解的 DHTML 功能和 JavaScript 编程，只用 HTML 就可以完成简单的 Web 应用。

图 5-20 显示的程序是仅用 HTML 实现的 ToDo 应用程序。将图 5-20 的内容保存在文件中。比如保存为/tmp/todo.html，那么 URL 就是 file:///tmp/todo.html。用 Web 浏览器读入的话显示出如图 5-21 那样的画面。

```

<html><head><title>ToDo</title>
<script type="text/javascript">
var todos = new Array()
function focusInput() {
    var itext = getElementById("itext")
    itext.value = ""
    itext.select()
    itext.focus()
}
function updateToDo() {
    var list = "<table>"
    for (i=0; i<todos.length; i++) {
        list = list + "<tr><td><input type='checkbox' "
onclick='checkToDo(" + i + ")'" + (todos[i][0]? " checked"> : ">
</td>\n")
        list = list + '<td><div>' + todos[i][1] + '</div></td></tr>'
    }
    list = list + "</table>"
    document.getElementById("list").innerHTML = list
    focusInput()
}
function addToDo() {

```

```

var input = document.getElementById("itext").value
if (input == "")return
todos[todos.length]= new Array(false, input)
updateToDo()
}
function addToDoTxt(event) {
    var kc = (event.keyCode || event.which);
    var v = document.getElementById("itext").value
    if (kc == 13 && v != "") {
        addToDo()
    }
}
function checkToDo(pos) {
    for (i=pos; i<todos.length-1; i++) {
        todos[i]= todos[i+1]
    }
    todos.length--
    updateToDo()
}
</script></head>
<body>
<ul>
<li>输入文字后按登录按钮
<li>完成后请点击复选框
</ul>
<input id="itext" type="text" value=""
onkeypress="addToDoTxt(event)"/>
<input type="button" value="登录" onclick="addToDo()"/>
<div id="list"></div>
</body>
</html>

```

图 5-20 只用 HTML 记述实现的 ToDo 应用



图 5-21 ToDo 应用的初始画面

在文字栏中输入预定项目，按下键之后追加项目，成为图 5-22 的样子。完成预定项目之后，单击复选框，消除项目。



图 5-22 在图 5-21 中追加了项目之后的样子

虽然这是个非常简单的程序，但它能把预定项目追加到列表里，完成之后可以消除掉，已经提供了作为 ToDo 应用所应具备的最基本功能。

可是，此应用仅仅把信息保存在 Web 浏览器的网页中，浏览器终止之后所有的信息就消掉了。重置页面也会消掉信息，所以基本上没有实用性。

5.2.7 和服务器的通信

使用 DHTML 之后，对于较简单的应用，在客户端就能够实现。但是，客户端无法保存数据，所以保存和获取数据时需要和服务器进行通信。

Ajax 是利用 XMLHttpRequest 对象来进行异步通信的。就像之前讲述的，不需要网页跳转，在后台就可以进行通信。

可是，在获取 XMLHttpRequest 的阶段，JavaScript 编程常常会碰到兼容性问题。

世界上有多种 Web 浏览器。能够支持 JavaScript 的有代表性的浏览器有：Internet Explorer、Firefox、Opera、Safari 和 Google Chrome 等。

而且，不同的 Web 浏览器获取 XMLHttpRequest 对象的方法也各不相同。因此，获取 XMLHttpRequest 对象时有必要像图 5-23 那样进行区分。

```
function GetXmlHttpRequest() {  
    var xmlhttp;  
    if(XMLHttpRequest) {  
        return new XMLHttpRequest();  
    }  
    else if (window.ActiveXObject) {  
        try {  
            return new ActiveXObject("Msxml2.XMLHTTP");  
        } catch (e) {  
            try {  
                return new ActiveXObject("Microsoft.XMLHTTP");  
            } catch (e) {  
                return false;  
            }  
        }  
    }  
}
```

图 5-23 取得 XMLHttpRequest 的 JavaScript 代码

如图 5-23 所示，问题在于 Internet Explorer 和其他的 Web 浏览器都不一样，非常麻烦。

5.2.8 使用Prototype.js的优点

使用 Prototype.js 的话，不需要那样麻烦的记述。

使用 Prototype.js 获取 XMLHttpRequest 对象的方法如下。

```
req = Ajax.getTransport()
```

和图 5-23 相比，这样就简单多了。另外，经常使用的 getElementById() 函数在 Prototype.js 中变成了 \$() 的形式，还有其他很多能让程序变短的技巧。

下面的程序示例使用了 Prototype.js。

5.2.9 在服务器上保存数据

现在看看如何在服务器上保存数据。在服务器上保存数据的 Ruby 程序如图 5-24 所示。

```
#!/usr/bin/env ruby
require 'cgi'
cgi = CGI.new
file = "todo.txt"
if cgi.key?("add")
  data = cgi["add"]
  open(file, "a") do |f|
    f.flock(File::LOCK_EX)
    f.print format("%d%d", Time.now.to_i, rand(10)), ":" # id
    f.print data, "\n"
  end
elsif cgi.key?("del")
  id = cgi["del"]
  open(file, "r+") do |f|
    f.flock(File::LOCK_EX)
    data = f.readlines.delete_if do |line|
      line =~ /^#{id}:/
    end.join
    f.rewind
    f.print data
    f.truncate f.pos
  end
else
  open(file, "r") do |f|
    f.flock(File::LOCK_SH)
    data = f.read
  end
end
print cgi.header("content-type"=>"text/plain", "charset"=>"UTF-8")
print data
```

图 5-24 保存数据的脚本示例

处理内容较简单：如果参数是 **add** 的话，就把内容追加到数据文件中；如果参数是 **del** 的话，就删除指定的项目；其他情况下，返回数据文件内容。

把图 5-24 的内容保存到 **data.rb** 文件中，与 **HTML** 文件保存在同一路径下。另外，如图 5-25 所示，需要在 **.htaccess** 文件中把 **.rb** 文件追加为

CGI 执行的设置。这里，HTML 文件名是 index.html。

```
Options +ExecCGI
AddHandler cgi-script .rb
DirectoryIndex index.html
```

图 5-25 在.htaccess 文件中追加的内容

现在来利用这个服务器端的脚本程序和 Prototype.js，在 ToDo 应用中追加保存数据的功能。

最初需要装载 Prototype.js。在 HTML 中 JavaScript 代码部分前面加入图 5-26 所示的 1 行内容。另外不要忘记 HTML 文件和 Prototype.js 需要保存在同一路径下⁴。

4 Prototype.js 的下载地址是 <http://www.prototype.org>。本文的脚本在 1.6.0.3 版本上完成了动作确认。

```
<script type="text/javascript" src="prototype.js"></script>
```

图 5-26 装载 Prototype.js 时的记述

下面将追加数据的加载和保存功能。从现在开始，使用 CGI 通过 HTTP 服务器进行处理。

具体地说，是将图 5-20 给出的 HTML 文件中 JavaScript 部分的 updateToDo()、addToDo()、checkToDo() 函数替换成图 5-27 中给出的同名函数。

```
function updateToDo() {
  new Ajax.Request("data.rb", {
    onComplete: function(req) {
      var lines = req.responseText.split("\n")
      todos = new Array()
      lines.each(function(line) {
        if (line != "") {
          var chunks = line.split(":")
          var id = chunks.shift()
          var item = chunks.join(":")
```



```

        todos.push(Array(false,item,id))
    }
})

var list = "<table>"
for (i=0; i < todos.length; i++) {
    list = list + "<tr><td><input
type='checkbox'onclick='checkToDo(\" + todos[i][2] +\")'\" +
(todos[i][0] ? \" checked>\" : \"></td>\n")
    list = list + '<td><div>' + todos[i][1] + '</div></td></tr>'
}
list = list + "</table>"
$("#list").innerHTML = list
focusInput()
}
})
}

function addToDo() {
    var input = $("#itext").value
    if (input == "") return
    var data = "add="+encodeURIComponent(input)
    new Ajax.Request("data.rb", {postBody: data})
    updateToDo()
}

function checkToDo(id) {
    var data = "del="+id
    new Ajax.Request("data.rb", {postBody: data})
    updateToDo()
}
}

```

图 5-27 使用了 Prototype.js 的数据加载和保存

图 5-27 中对功能进行改善的本质内容是各函数中出现了一次 `Ajax.Request`。 `Ajax.Request` 的使用方法如下所示。

```
new Ajax.Request(url, options)
```

`url` 是请求的对象地址，实际的动作由 `options` 中的指定的哈希值来控制。 `options` 中可以指定的内容如表 5-4 所示。

表5-4 `Ajax.Request`的`options`指定的内容

--	--

名 字	说 明
asynchronous	是否同期（true/false），缺省是true
method	请求的方法（“get”/“post”），缺省是“post”
onComplete	数据获取完毕时调用的函数，参数是XMLHttpRequest对象
onException	发生异常时调用的函数
onFailure	连接失败时调用的函数
onSuccess	连接成功时调用的函数
parameters	传给请求对象的参数（字符串）
postBody	POST时的内容体（字符串）

ToDo 数据加载时（图 5-27 的 `updateToDo` 函数），指定了 `onComplete` 函数，这样读完数据之后将调用指定的函数。JavaScript 可以简单地编写匿名函数⁵，所以不用总是考虑函数名，比较方便。此函数按如下的顺序进行处理：

5 匿名函数是指没有指定名字的函数。JavaScript 通过使用未命名的 `function` 语句，可以不用定义函数，而把函数当做值来使用。

1. 取出 `XMLHttpRequest` 对象的 `responseText` 属性中读取的数据（字符串）；
2. 将数据按行分割，更新 `todos` 变量。Prototype.js 提供了 `split`、`each` 等 Ruby 类的方法，很有用；
3. 调用 `updateToDo()` 函数，更新 ToDo 列表。

上面的处理是异步进行的，就算获取数据花费了很长时间，ToDo 应用在数据为空时也能够执行动作，当数据全部读入时再更新画面。

5.2.10 Web应用的脆弱性

实际上，这个 ToDo 应用还有不少麻烦问题，比如 XSS（跨站点脚本）问题。像现在的代码，输入 ToDo 的项目时如果内容里包含有 HTML 标签的话，就会解释成 HTML 内容。一个人使用的情况下还没

什么问题，若是不确定的多数人使用 Web 应用的话，不仅可能会随便加入链接、图像等，还有可能导致使用 JavaScript 的深层问题。

为了避免这样的事情发生，有必要将 HTML 标签转换为其他安全的表示文字。这里最简单的方法是利用 Prototype.js 的 HTML 转义功能，具体是把函数 `updateToDo()` 中图 5-28 中 (A) 的部分改正成 (B) 那样。

像这样，对于各种人都使用的 Web 应用而言，有很多地方都需要特别注意。

```
(A)
list = list + '<td><div>' + todos[i][1] + '</div></td></tr>'

(B)
list = list + '<td><div>' + todos[i][1].escapeHTML() + '</div></td></tr>'
```

图 5-28 `updateToDo()` 函数的 XSS 对应：把(A)部分改写成(B)

5.2.11 使用JavaScript的感觉

笔者平常没有使用 JavaScript 编程。为了这次的讲解，相当下功夫地进行了学习。对于我来说，已经好久没用 Ruby 以外的动态语言进行编程了，觉得非常有趣。既然学了，在这里就总结一些使用 JavaScript 的感觉。

作为动态语言名副其实

因为有闭包这样的匿名函数，可以较简单地实现 Ruby 里的块状操作。尽管像 `function` 这样的专用语太长，感觉有点麻烦。前面介绍过继承的实现有些繁杂，但 JavaScript 编程一般不会出现继承，所以没有在意。

DHTML比想象的更有趣

把 HTML 作为数据进行操作，改变属性或是追加节点等都相当有趣。特别是可以直接看到操作的结果，感觉很直观。

Prototype.js也不错

Prototype.js 能够把单独使用 JavaScript 编程时繁杂的部分给屏蔽掉。作为 Ruby 编程语言的作者，我也很高兴 Prototype.js 包含了 Ruby 里的 Enumerable 和 String 功能。

调试比较麻烦

JavaScript 里就算有程序错误，Web 浏览器也不会显示任何信息。想要确认程序的状态，只能多次使用 `alert()`。`alert()` 可以弹出一个消息窗口，显示作为参数接收的字符串。类似于 Ruby 程序中使用 `print` 的调试方法。

Firefox 提供了 Firebug 的扩展功能，对于 JavaScript 的调试非常有用。在本书这次的写作中也起到了很大的帮助。

兼容性的问题

各个 Web 浏览器相互独立地实现了 JavaScript 解释器。虽然 JavaScript 有标准规格(ECMA-262)，但在动作细节上各有不同（特别是 Internet Explorer）。这次介绍了的获取 XMLHttpRequest 对象的方法就是一个例子。

我有一位朋友的业务是做 Ajax 应用开发。按照他的说法，Ajax 开发最难的一点就是 Web 浏览器间 JavaScript 的兼容性问题。不同的 Web 浏览器都各自独立实现了 JavaScript 的处理程序。虽说有标准规格，细节部分很多都不兼容。在 Firefox 和 Opera 上能够执行，在 Internet Explorer 上却无法执行，类似情况经常出现，让人十分头疼。饮水思源，大家在使用 Ajax 应用时，也许应该想起那些因为兼容性问题而无法入眠的技术人员。

名字的重要性

在美国，人们相信一个说法：所有人和物都有名有姓，知道了他的名字就可以控制他。因此，他们会对自己的真实姓名保密，仅仅对家人或是真正能够信赖的人才公开。对外只告诉译名。这么一说，厄休拉·勒古恩的《加德战记》就是这么做的呢。“加德”是主人公的真实名字，在故事中基本没有使用，一般总叫他为“鹧子”。

有时候会觉得这种说法在某种程度上确实有道理。也就是说，事物名字具备一种说不出原因的神秘支配力量。

比如，本章讲解的 **Ajax** 也是这样。**Ajax** 本来只是多种技术的组合，而且没有什么新奇之处，只是自然地组合在一起。但是，**Ajax** 却风靡一时，改变了之后的网站形象。最近没怎么听到关于 **Ajax** 的说法，主要是因为网站使用 **Ajax** 已经被认为是常识性的问题。

我个人感觉，**Ajax** 成功的原因不仅仅因为单纯的技术组合，还在于名字有魅力。名字确实很重要。

Ruby 也有同样的感觉。我在 1993 年开始开发 **Ruby**，仿造 **Perl** 取了 **Ruby**（红宝石）的名字。正是因为这个短而美丽的名字，才能够维持长时间的开发动力，让如此多的用户对 **Ruby** 语言感兴趣。

因此，我设计上的一个座右铭就是名字很重要。设计任一功能时，我首先会着重考虑它的名字。在我作为编程人员的生涯中，名字好的功能都成功了，而名字不太好的功能事后往往让人后悔。

实际上，对于有人提出追加 **Ruby** 功能的要求也经常看名字。拒绝时的回答是：“我们明白这个要求。知道若有此功能会更方便。但是不喜欢这个名字。若有一个好名字我们会采用。”而且，对于因为不喜欢名字而没有实现的功能，之后我也没有觉得后悔。

也许可以这么来考虑吧。起了一个合适的名字本身就意味着功能设计得正确。反过来，起了不好的名字说明设计者自己也没有完全理解应完成什么样的功能。我个人认为，为一个功能起个合适的名字就已经完成了八成的工作。

请大家试试起名字更加用心些会如何？也许下一个项目成功的秘密就在于此呢。

第 6 章 **Ruby on Rails**

6.1 MVC 和 Ruby on Rails

MVC 是设计 GUI 程序时的设计模式¹ 之一。名字来自于模型（Model）、视图（View）和控制（Controller）三个单词的首字母。大部分设计模式仅决定程序某一部分的构成，而 MVC 决定了应用程序的整体部分，有时候也被称为架构模式。

1 设计模式是软件设计，特别是在以面向对象为基础的软件设计里经常使用的编程用语。最简单的例子就是 for 循环。

MVC 最早是作为编程语言 Smalltalk 带窗口（GUI）的应用架构指针而诞生的。据说，发明 MVC 的是当时美国施乐公司下属的帕洛阿尔托研究中心（PARC）的挪威人 Trygve Mikkjel Heyerdahl Reenskaug²。Simula 的开发者是 Kristen Nygaard（挪威），C++的开发者是 Bjarne Stroustrup（丹麦），在面向对象开发的历史上，北欧人贡献卓著³。

2 Reenskaug 发明 MVC 大约是在 1978 年。最初被叫做 MVCE

（Model·View·Controller·Editor），<http://heim.ifi.uio.no/trygver/themes/mvc/mvc-index.html>。

3 每年都在丹麦举行 Java 和面向对象的会议 JAOO。我和他们 3 人就是在那里认识的。我参加过两次这个会议，每次都收获颇丰。只可惜离日本太远，出席会议不太方便。

6.1.1 模型、视图和控制的作用

MVC 中的模型，是表现窗口中表示内容（信息）的对象。模型代表的只是信息（名字、数值等抽象的信息），它不能包含如何来显示这些信息的信息。

视图，代表将模型中包含的信息在窗口中进行表示的对象。视图知道要表示的模型的信息，而模型一般不知道要表示自己的视图信息。

控制，是从用户端接受输入，对视图和模型进行操作的对象。

以上关系如图 6-1 所示。用户通过鼠标和键盘把输入传送给控制部分。控制部分可以调用模型和视图，根据需要调用相应的方法，对应用程序整体进行控制。

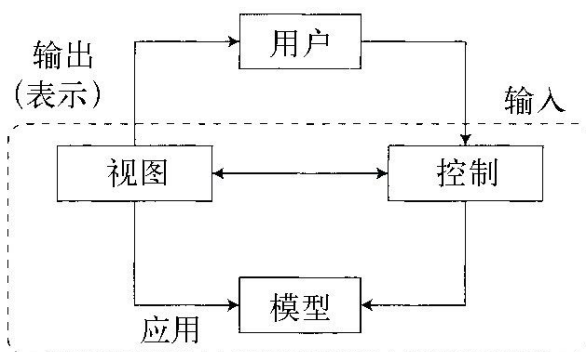


图 6-1 模型、视图以及控制各部分的功能和相互间的关系

视图可以访问模型，在窗口中显示模型的状态。查询模型的状态有多种方法。**Smalltalk** 一般是利用 **Object** 类中内嵌的 **Observer** 模式⁴。还有所谓的轮询法，每隔一定时间，在发生显示事件时去查询模型的状态。

4 **Observer** 模式，是一种降低多个对象间依赖性的较有效的设计模式。使用目的是观察状态的变化，并发出相应的通知。

视图部分能否访问控制部分要视不同应用而定。图 6-1 表示的是能够访问的情况。对于简单的窗口应用，在大多数情况下，视图没有必要去访问控制。但是，当应用变得复杂时，视图和控制之间的相互作用就变得越来越频繁，很多时候也就变成一体了（以后会讲到）。

另一方面，模型没有必要设计成可以访问视图部分或控制部分。从图 6-1 可以看到，没有从模型出来的箭头，这是因为模型和视图有可能不是一对一的关系。换句话说，同样的模型可以对应多个视图。

比如，考虑一下钟表应用。模型相当于记录时间的机构。视图则是实际表示时间的部分，既可以是模拟表，也可以是数字表。这种情况下，只要更换视图部分的对象就能实现钟表的不同表示。因此，如果把模型设计成直接访问视图就不方便了。

把代表应用逻辑的模型部分与提供界面的视图和控制部分分离开来还有其他一些好处。将来，当更换界面或是追加其他一些大规模变更时，对应用逻辑的影响也较少。相对应用逻辑变更来说，界面变更的情况占绝大多数，所以把它们分离开来是很合理的。

6.1.2 用秒表的例子来学习MVC模式

只作理论上的讲解，还不容易让人理解使用 MVC 有什么方便。这里，我们准备了示例程序。

这次使用的例子是一个简单的秒表程序。画面上表示出经过的时间，按键盘上任意键启动或停止秒表，按下 q 键时退出程序。计时单位和普通的数字秒表一样，是 1/100 秒。

为了以 MVC 模式来实现程序，首先开始考虑对象的构成。既然是 MVC 模式，需要准备模型、视图和控制这 3 个对象。

如表 6-1 那样简单地进行划分。这是一个典型的 MVC 构成。类名中直接包含了代表其作用的单词，可以看到重要的是不能混淆它们的功能。

表6-1 秒表程序中MVC模式的构成

	处理内容	类 名
模型	时间计算	WatchModel
视图	表示	WatchView
控制	用户进行的操作	WatchController

那么，先来看看模型类 WatchModel 的代码（参见图 6-2）。

```
#!/usr/bin/ruby

require "observer"

# the Model class count clock ticks
class WatchModel
  include Observable
  def initialize
    @running = false
    @time = 0
    @last = 0.0
    Thread.start do
      loop do
        sleep 0.01
        if @running
          now = Time.now.to_f
          @time += now - @last
          @last = now
        end
      end
    end
  end
end
```



```

        changed
        notify_observers(@time)
    end
end
end
end

def start_stop
    @last = Time.now.to_f
    @running = ! @running
end
def time
    @time
end
end
end

```

图 6-2 秒表程序中的模型部分

首先，为了实现模型和视图的协作，本程序利用了设计模式之一的 Observer 模式。所以，程序的开头要加载 **observer** 库。接着，因为 **WatchModel** 要成为 Observer 模式的观察对象，所以在类定义的开头要包含 **Observable** 模块。

类 **WatchModel** 有 3 个方法：**initialize**、**start_stop** 和 **time**。**initialize** 初始化模型，让秒表开始计数。**initialize** 方法中使用的无限循环有点特别。但是，在使用线程时程序是可以写成这样的。请注意，此循环是在别的线程中进行的。

在主循环中，每次 **sleep 0.01** 秒。这次采用的计时精度是 1/100 秒，所以每次加算 0.01 秒（1/100 秒）。在实例变量 **@running** 为真的时候进行计时，状态变更的时候则通知一直在观察此模型的观察者（这里是视图对象）。具体执行是利用了设定变更标志的 **changed** 方法和实际通知的 **notify_observers** 方法。

start_stop 方法用来设置例程变量 **@running**，对秒表的开始和停止进行切换。就好像实际的秒表那样执行。**time** 方法则返回经过的时间。

initialize 进行初始化（启动内部的主循环），**start_stop** 启动或停止秒表，用 **time** 方法获取到现在为止经过的时间。这就是秒表

的应用逻辑。若还非要再列举的话，也许还需要复位功能。但是，通过这些方法，**WatchModel** 类基本上提供了作为秒表应用程序模型所必须具备的功能。

6.1.3 生成视图和控制部分

模型的下一步就是视图，图 6-3 给出了秒表程序的视图部分。构成视图的 **WatchView** 类很简单。初始化对象的 **initialize** 方法接受表示对象的模型作为参数，使用 **add_observer** 方法把自己登录为模型的观察者。因为使用了 **observer** 库，只有这么一程序，就可以实现在模型发生变更的同时调用视图对象的 **update** 方法。**update** 方法只是简单地将到现在为止的累积时间表示出来。在这里为了控制画面，使用了 ANSI 转义符（ESC [8D），让光标移动到行的开头再显示时间。

```
# the View
class WatchView
  def initialize(model)
    model.add_observer(self)
  end
  def update(time)
    printf "\e[8D%02d:%02d", time.to_i, (time-time.to_i)*100
    STDOUT.flush
  end
end
```

图 6-3 秒表程序的视图部分

最后是生成控制类（参见图 6-4）。有多种方式能够生成模型、视图和控制这 3 个对象间的关联。这里为了构成简单，让控制类打头，然后生成另外两个对象，把三者进行了关联⁵。

5 复杂的应用使用所谓的 DI（Dependency Injection）方法比较好。

```
# the Controller
class WatchController
  def initialize
    @model = WatchModel.new
    @view = WatchView.new(@model)
    system "stty cbreak -echo"
```

```
begin
  @view.update(@model.time)
  loop do
    break if STDIN.getc == ?q
    @model.start_stop
  end
  ensure
    system "stty cooked echo"
  end
end
end
end
WatchController.new
```

图 6-4 秒表程序中的控制部分

控制类 `WatchController` 的 `initialize` 方法生成了模型和视图对象，让三者发生了关联。

控制类接受用户的键盘输入。这里不想使用 `curses` 库那样大规模的东西，所以用 `system` 方法调用 `stty` 命令对终端进行设定。

`stty cbreak-echo` 设定成让终端对每一个键的输入都进行响应（`cbreak` 模式），不显示输入的文字（`-echo`）。这样程序就可以直接接受键盘的输入了。

程序结束时希望回到初始状态，所以执行 `stty cooked echo` 命令，回到通常设定，即以行为单位读取（`cooked`）输入并显示输入的文字（`echo`）。使用 `ensure` 来保证程序终止时一定回到以前的设定状态。

剩下的就是控制部分的主循环。从键盘读取每个输入，如果是 `q` 则结束，否则就调用模型的 `start_stop` 方法，对秒表进行开关控制。

6.1.4 GUI工具箱与MVC

MVC 是 Smalltalk 语言用来作为一般 GUI 应用的构成方式。但是，其他的语言没有广泛使用 MVC。那么，一般的 GUI 工具箱一定也有某种构造模式。对于它们来说，模型、视图和控制都是如何构成的呢？我们来和 MVC 作一个比较。

多数的 GUI 工具箱包括部件（component）、事件（event）和回调（callback）。这里拿一个使用了 Ruby/Tk 的 GUI 应用为例（参见图 6-5）。

```
# Ruby/Tk
require "tk"
TkButton.new(nil, 'text' => 'hello',
              'command' => proc{print "hello\n"}).
  pack('fill' => 'x')
TkButton.new(nil, 'text' => 'quit',
              'command' => proc{exit}).pack('fill'
              => 'x')
Tk.mainloop
```

图 6-5 使用 Ruby/Tk 的单纯的 GUI 应用

执行图 6-5 的代码会显示如图 6-6 那样的一个小窗口。按下上面的 hello 键，标准输出会输出 hello，按下下面的 quit 键则退出程序。



图 6-6 图 6-5 程序的输出结果

在这个程序中，功能相当于视图和控制的是 **TkButton** 类。

TkButton 类中有（1）GUI 按钮作为显示效果，（2）用户按下按钮（在按钮上点击鼠标）来实现功能。因为使用了按钮这样的隐喻方式，GUI 应用不需要说明书就能够操作，非常易懂。这样，外表和功能紧密地结合在一起。

在典型的 GUI 应用程序中，鼠标光标移动到按钮上时改变按钮的颜色或视觉等效果（看起来凸出来），在点击的瞬间使按钮凹下去等，控制和视图紧密协作，共同实现这些功能。在这种密切相关的情况下，对类进行分割的做法并不明智。所以，多数的工具箱都是视图和控制部分相结合而被称为部件（component）。按钮、菜单或菜单条等都是典型的 GUI 部件。

如果说视图和控制结合成为了部件，那么和模型部分相当的又是什么呢？

实际上，GUI 工具箱没有提供和模型相当的东西。作为替代而提供了访问模型的入口，这就是回调。

图 6-5 的程序中有两处调用了 `proc` 方法，这就是回调。`proc` 是一种方法，可以将完成一定操作的程序块作为对象调出来。当用户输入发生特定的事件时（按下按钮等），GUI 部件通过回调调出功能程序块。

回调一般是较小的操作功能，发挥“启动”模型对象的功能。回调的功能就好像是黏结“视图+控制”组成的部件和模型之间的“胶水”。

从以上可以看到，工具箱中各个部件和 MVC 类似的构造可以按照如下来理解。

V+C M	GUI 部件 回调
----------	--------------

6.1.5 同时使用工具箱和MVC

GUI 工具箱是靠这样的部件组合来开发应用的。那么，使用了工具箱的 GUI 编程不能使用 MVC 吗？

GUI 工具箱提供了个别的部件，但对应用整体的架构基本上没有支持。就算使用了 GUI 工具箱，把整体应用构造成 MVC 模式，这样就可以开发出系统构成明确是容易维护的应用了。

那么，让我们来生成一个使用 GUI 工具箱，并采用 MVC 构成的应用吧。具体来说，就是把前面的秒表进行视图化（GUI 化）。

图 6-7（`mvc-gui.rb`）是使用 Ruby/Tk 和 GUI 化的秒表程序。图 6-7 的程序执行之后会显示出图 6-8 那样的窗口。按下 `start/stop` 按钮后，秒表开始计时，再次按下则停止。按下 `quit` 钮则终止程序。

```

#!/usr/bin/ruby

require "observer"
require "tk"

# the Model class count clock ticks
class WatchModel
  include Observable
  def initialize
    @running = false
    @time = 0
    @last = 0.0
    Thread.start do
      loop do
        sleep 0.01
        if @running
          now = Time.now.to_f
          @time += now - @last
          @last = now
          changed
          notify_observers(@time)
        end
      end
    end
  end

  def start_stop
    @last = Time.now.to_f
    @running = ! @running
  end

  def time
    @time
  end
end

# the View+Controller
class WatchWindow
  def initialize
    model = WatchModel.new
    model.add_observer(self)

    @label = TkLabel.new(nil).pack('fill'=>'x')
    self.update(0)
    btn = TkButton.new
    btn.text('start/stop')
    btn.command(proc{model.start_stop})
    btn.pack('fill'=>'x')
    btn = TkButton.new
    btn.text('quit')
  end
end

```

```

        btn.command(proc{exit})
        btn.pack('fill'=>'x')
        Tk.mainloop
    end
    def update(time)
        @label.text format("%02d:%02d",
                           time.to_i,
                           (time-time.to_i)*100)
    end
end
WatchWindow.new

```

图 6-7 秒表进行 GUI 化之后



图 6-8 图 6-7 程序的输出结果

一起来看看图 6-7 的程序内容。模型的 **WatchModel** 和以前的文字版完全一样。**MVC** 构成把应用的本质部分分离出来，所以应用的画面变化基本上不会给模型带来什么影响。

剩下的视图和控制，是把 **GUI** 工具箱及事件处理二者结合起来。模型和视图作为不同对象分割开的意义现在还看不出来。这里我们把模型和视图二者的功能用 **WatchWindow** 类统一起来。

类 **WatchWindow** 使用 **Observer** 模式可以收到模型的变更通知，这点和文字版的视图是相同的。收到变更通知后会调用 **update** 方法，它改变 **label** 对象的文字来表示时间。

文字版的控制是对用户的输入进行响应。图 6-7 则是交给了 **Ruby/Tk**。**initialize** 方法中配置了 **GUI** 部件（表示时间的标签和秒表的操作按钮），指定了对模型对象进行操作的回调。文字版里的循环部分由 **Ruby/Tk** 的启动事件循环的 **Tk.mainloop** 来实现。

这样做了之后，这个应用把多个 **GUI** 部件的小 **MVC** 统一成一个 **GUI** 应用的大 **MVC**，实现了多层构造。

因此，使用了 GUI 工具箱的应用也可以利用 MVC 构成。采用了这样的构成之后，和功能本身相关的变更就只针对模型部分来执行，和用户界面相关的变更就只针对视图和控制来执行即可。

6.1.6 MVC的优缺点

上述 MVC 对于软件设计有很多优点，总结如下：

可以更换界面

刚才已经表明，对于模型未作任何变更，文字版的秒表变成了 GUI 版程序。采用 MVC 构造可以容易地更换界面。

一个模型对应多个视图

使用 MVC 构造，不仅可以更换界面，一个模型可以同时赋予多个界面。比如和数字表同时执行的模拟表；在表格计算的同时，和数值相关的图形也一起联动等。使用 MVC 构造之后，能够自然地实现多个复杂的视图。

多个视图可以同时响应

使用 MVC 构造准备好多个视图时，只要编程不出问题，所有的视图都会同时响应模型的状态。比如，在表格计算时，每格数值变化的瞬间，图形也随之发生变化。

容易测试

MVC 将应用的本质内容作为模型独立出来，在生成界面之前可以对应用逻辑进行测试。

另一方面，MVC 不是没有缺点。

复杂性

将应用分割成模型、视图和控制，各个部分设定各自的功能，对变更要无滞后地通知等，要做到这些需要复杂的处理。不过，像这次的例

子程序这样仅用 60 行代码就完成了。只要使用适当的语言和适当的库，可以将复杂性尽量降低。

强关联性

虽然分离为不同的对象，但模型、视图和控制三者相互之间保持了强关联性。对模型对象进行了功能追加这样的变更之后，相应地也必须对视图和控制进行变更。

6.1.7 Web应用中的MVC

到此为止，我们见识了 GUI 应用中的 MVC 案例。那么，在 Web 应用中，MVC 是如何发挥作用的呢？

首先，需要意识到 Web 应用的基本是 HTTP。HTTP 的一次处理经过了下面 1 到 3 的过程：

1. Web 浏览器对应于用户的操作，向 Web 服务器发出 HTTP 请求。
2. Web 服务器根据请求，准备好发送到 Web 浏览器的数据。
3. Web 服务器把数据以 HTTP 响应的形式送还 Web 浏览器。

用 MVC 模式来对应上述的过程应该是以下形式。

1. Web 浏览器发送来的 HTTP 请求通过 Web 服务器传给控制部分。Web 应用框架的分配器（分配部件）把请求传递给合适的控制部分。
2. 控制部分操作的模型和请求的信息相对应，同时指定显示使用的视图。视图从模型启动，一边引用模型一边准备发送给 Web 浏览器的数据。
3. Web 服务器把数据以 HTTP 响应的形式送还 Web 浏览器。

和较复杂的 GUI 应用不同的是，HTTP 的控制的流程对于 1 个请求会返回 1 个响应，从控制到视图的处理顺序是明确的。

图 6-9 显示了作为使用 MVC 的 Web 应用的代表，Ruby on Rails 的 MVC 构成。不过，虽说 Rails 采用了 MVC，但它的用法和传统的 MVC 模式还是有些微妙的不同。

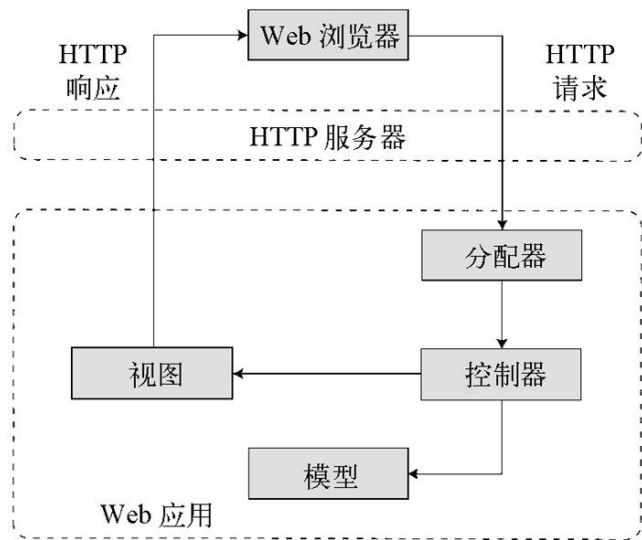


图 6-9 Web 应用中的 MVC，Ruby on Rails 的情形

Rails 中也存在模型、视图和控制，各自保存在独立的目录下。构成 Rails 应用的目录下有一个叫 `app` 的子目录，在它之下又配置有以下的目录。`helpers` 是指帮助程序。

<code>app/models</code>	模型
<code>app/views</code>	视图
<code>app/controllers</code>	控制器
<code>app/helpers</code>	帮助程序

到此为止，在说明的 MVC 模式中，模型是指业务逻辑，控制器指输入控制，视图指输出控制的功能。Ruby on Rails 上的 MVC 各部分功能稍有不同。Rails 中的模型相当于数据库层，视图指显示用的模板，控制器指控制用的类（包含了应用逻辑）。

传统的 MVC 模式和 Rails 的 MVC 模式相比较的话，大体上如表 6-2 所示的对应关系，相互之间有些不同。

表6-2 传统的MVC和Rails MVC的比较

--	--

传统的MVC	Rails的MVC
模型	模型+控制器
视图	视图
控制器	分配器+控制器（的一部分）

本来，Web 应用广泛采用 3 层的系统架构。所谓 3 层是指用户界面（UI）层、业务逻辑层和数据库层这 3 层。传统的 MVC 模型把业务层和数据库层合为一体称为模型。这和 SmallTalk 不太使用外部数据库（系统本身有持久化功能）也许有一定的关系。

另一方面，由于 HTTP 特性的关系，那些把 UI 部分的复杂性全部交给 Web 浏览器来完成的 Web 应用，相对来说 UI 层会变弱。控制器用一般的设计就足够了，模型和视图间的关联也不必要了。因此，把模型分割成数据库层和业务逻辑层，下层称为模型，上层称控制器。

实际上，Rails 的控制器不仅是业务逻辑，也还担当了一部分控制功能，不能说完全是名不副实。

同样是 MVC 模式，各自的功能却有些许不同，感觉有点奇怪。也许是由历史原因造成的。

词语的意思也稍稍有些不同，有点麻烦。不过，Rails 采用与传统 MVC 不同的功能划分不能说不好。Rails 支持的以数据库为中心的应用架构中，与把业务逻辑和数据库操作合为一体称作模型相比较，更清楚地分割开来，会更容易维护，所以才出现了 3 层模型。

6.2 开放类和猴子补丁

Wikipedia 英语版对猴子补丁（monkey patch）的解释是，在动态语言中，不改变源代码而对功能进行追加和变更。

历史上，Zope（Python 的 Web 框架）中，修正别人的程序错误时在程序后面追加更新的部分，叫做“杂牌军补丁”（guerilla patch），杂牌军 → 猩猩（gorilla）→ 猴子（monkey），不知道是怎么联想的。本来的含义是替换已有的方法（打补丁），现在灵活使用开放类，变更和追

加方法全部称为猴子补丁。这里也许有“打补丁的对象是类”这一层意思。

6.2.1 开放类

Ruby 的类的特征是所谓的开放类，相对其他多种语言而言，特别易于打猴子补丁。

开放类是指定义了之后也能任意追加内容的类。Ruby 中，普通的类是按以下方式定义的。

```
class Foo < Bar
  def plus2(arg1, arg2)
    return arg1 + arg2
  end
end
```

上面的代码新定义了一个叫 **Foo** 的类。它的父类是 **Bar**。类里用 **def** 语句定义了 **plus2** 的方法。

Foo 类可以继承父类 **Bar** 拥有的方法，**Foo** 类的对象可以使用 **Bar** 类的方法和 **plus2** 方法。

像这样，Ruby 可以定义新类，也可以给已有的类追加定义。下面就是给已经定义了的类 **Foo** 追加方法。

```
class Foo
  def times2(arg1, arg2)
    return arg1 * arg2
  end
end
```

有了上面的代码，**Foo** 类就可以使用从 **Bar** 类继承来的方法，以及前面定义的方法 **plus2** 和新定义的方法 **times2**。这样对 **Foo** 类重新定义之后，即使是已经生成的 **Foo** 类对象，也拥有新增加的功能。

Ruby 中，可以把 `String` 类、`Array` 类等基本的数据类型及所有的类都作为开放类处理，可以自由地追加功能。

6.2.2 猴子补丁的目的

使用开放类，不改变原先的代码，替换方法（相当于打补丁）被称为猴子补丁。使用此技术可以完成以下功能。

功能追加

利用开放类可以给已有的类追加功能。向标准的字符串和数组类追加方法很大胆，有时候就非常有效。

功能变更

不仅是追加方法，替换方法可以让已有的类发挥完全不同的功能。比如，`Math` 库作了下面的功能变更。

Ruby 里，整数相除的结果还是整数：

```
a = 1/3
```

通常的结果是舍去 $1/3$ 成为 `0`。不过，如果加载 `Math`， $1/3$ 的结果可以返回有理数 $1/3$ 。另外也可以扩展到求负数的平方根返回复数值。

修正程序错误

因为重新定义了有程序错误或有副作用的方法，不用修改原来那部分的代码就可以解决问题。这也是本来的（称为杂牌军补丁的时候）猴子补丁的目的。

钩子

有时候想在每个方法调用的同时增加一些其他处理，比如日志输出功能。这种伴随方法调用而进行的处理称为“钩子”（hook），钩子的追加也可以用猴子补丁来实现。

缓存（cache）

在计算量很大，而且一次计算之后结果可以反复使用的情况下，在一次计算完成后，对方法进行替换可以提高处理的速度。比如在 `ate` 库中，罗马儒略历（`Julius` 历）的原点的计算就用了此方法。

6.2.3 猴子补丁的技巧

可以把 `Ruby` 提供的对方法、类和模块进行操作的功能（参见表 6-3）运用到打猴子补丁上。最基本的功能就是给已有的方法改名或取消。

表6-3 对方法、类和模块进行操作的功能

名 称	功 能
<code>alias</code>	给方法另起别名
<code>include</code>	加入其他模块中的方法
<code>remove_method</code>	取消本类中的方法
<code>undef</code>	取消方法

`undef`

`undef` 有把方法取消定义的功能。用 `undef` 不仅可以取消本类中的方法，也可以取消父类中定义的方法（参见图 6-10）。



图 6-10 用 undef 来取消方法

这里使用了 `undef` 语句。`undef_method` 方法也可以提供同样的功能（参见图 6-11）。语句中用符号指定要取消的对象的方法名。符号是指跟在冒号后的名字，Ruby 用这种方式表示程序中的名字等。

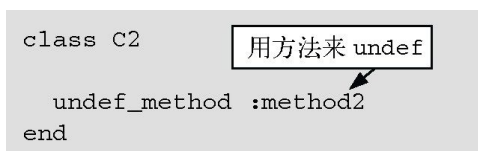


图 6-11 用 undef_method 来取消方法

`undef_method` 是类（正确地说应该是父类的模块）中的方法，可以在类定义或是模块定义中来调用。

和 `undef`、`undef_method` 非常相似的功能还有 `remove_method`。和 `undef` 不同的是，`remove_method` 不是语句，只是提供了方法。

现在来取消类中的方法。如果取消的对象不是本类中定义的方法则会出错（参见图 6-12）。而且，`remove_method` 仅是取消本类中定义的方法，如果父类中定义有同名的方法，则今后会直接调用父类的那个方法。

```
class C1 < Object
  def method1
    p :method1
  end
  def method2
    p :method2
  end
end

class C2 < C1
  def method2
  end
  def method3
  end

  remove_method :method3

  remove_method :method2

  remove_method :method1
end
```

类 C1 提供的方法

类 C2 提供的方法

删除 C2 的 method3

删除 C2 的 method2
C1 的 method2 就有效了

C2 中没有 method1, 出错

图 6-12 用 `remove_method` 删除方法

alias

给方法起一个别名（参见图 6-13），起了别名的方法和以前的方法只是名字不同，功能完全一样。不过，用 `undef` 或 `remove_method` 删除了以前的方法之后，别名的方法还能使用（参见图 6-14）。


```
class C3 < Object
  def foo
    puts "foo"
  end
  alias bar foo
end

c3 = C3.new
c3.foo # 输出"foo"
c3.bar # 同样输出"foo"
```




图 6-13 用 `alias` 给方法起别名

```
class C3 < Object
  def foo
    puts "foo"
  end
  alias bar foo
  undef foo
end

c3 = C3.new
c3.bar # 输出"foo"
c3.foo # 已经被 undef, 所以出错
```




图 6-14 用 `alias` 给方法起别名

`alias` 是 Ruby 的语句。就像 `undef` 有方法版的 `undef_method` 一样，`alias` 也有方法版的 `alias_method`。`alias` 是打猴子补丁时最常用到的功能。一般做法是，给某个方法用 `alias` 起个别名，在重新定义的方法中用别名来调用原来的方法，从而给原来的方法增加新功能。图 6-15 显示了调出方法后用猴子补丁来增加日志功能的例子。

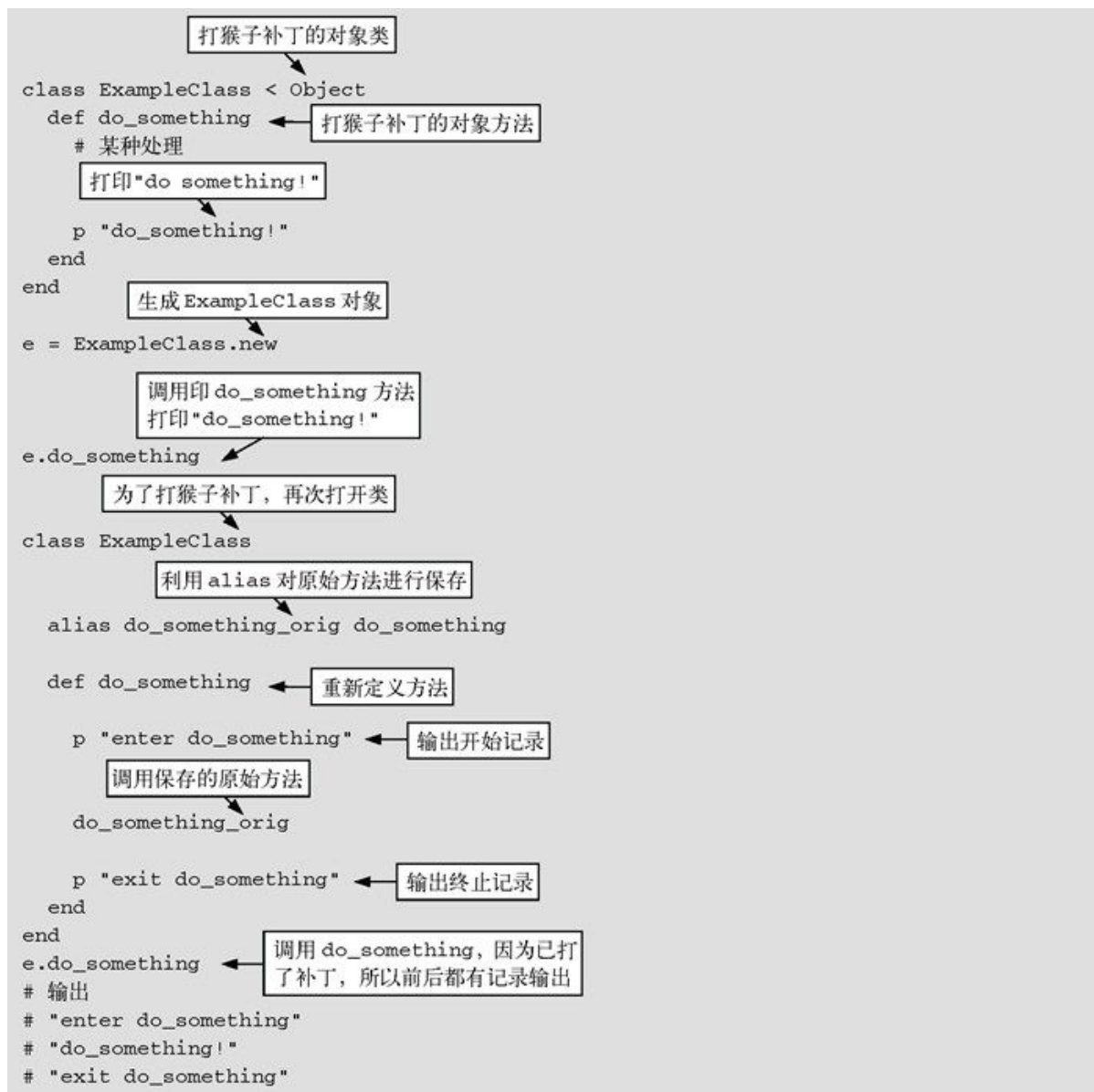


图 6-15 用 alias 打猴子补丁的例子

include

include 是类或者模块中把其他模块的方法包含进来的功能。它不像 **undef** 或 **alias** 那样直接操作方法，而是用开放类来给已有的类追加模块功能。

这时候需要注意的是，用 **include** 包含进来的模块，是作为（假想）父类来处理的，所以模块提供的方法在优先顺序上要低于本类提

供的方法（参见图 6-16）。用 `include` 的时候还要注意方法名的重复问题。

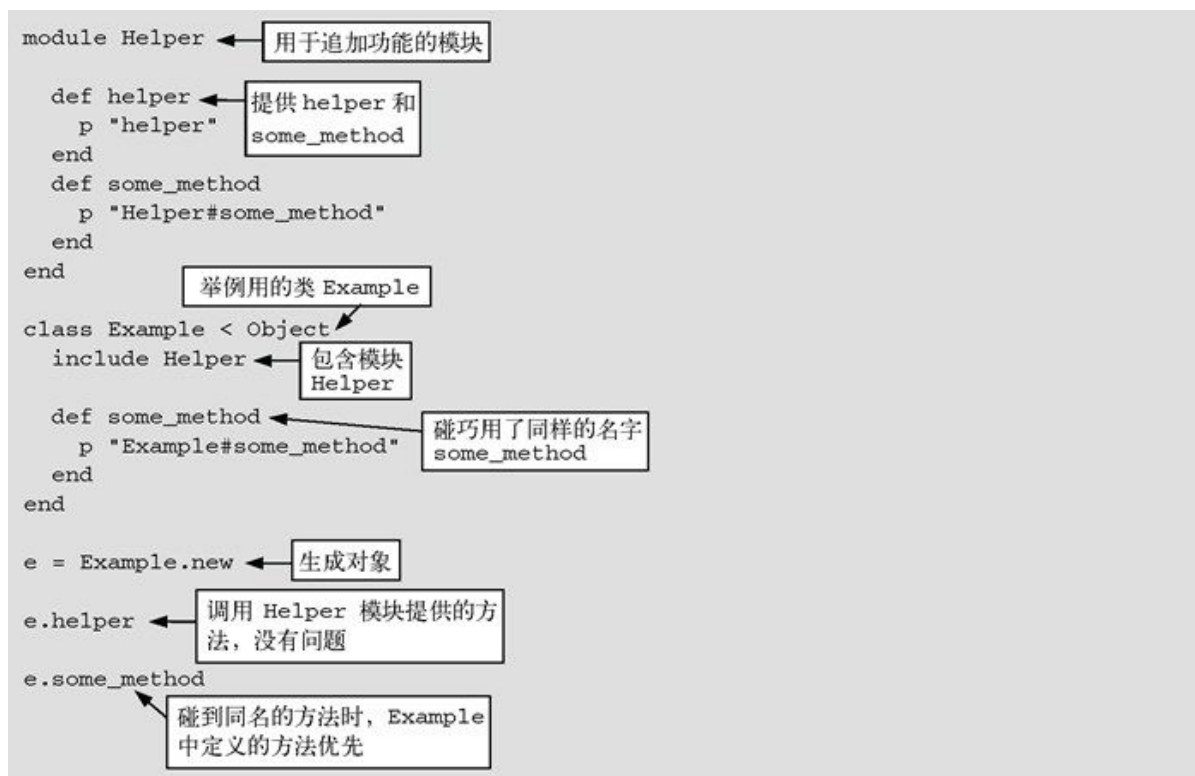


图 6-16 `include` 涉及的方法的优先顺序

6.2.4 灵活使用开放类的库

作为灵活使用开放类库的例子，这里来介绍 `jcode`。

Ruby 1.8 的字符串一般都是字节的组合¹，`str[n]` 将返回字符串的第 `n` 个字节。如果文字是用 EUC 或 JIS 这样的多字节编码表示的话，除非使用正则表达式²，否则无法对应多字节文字。

¹ Ruby 1.9 扩展了字符串，成为了文字的组合。详情见第 7 章。

² Ruby 的正则表达式能够对应多字节，只要适当地指定了文字编码，就可以处理 EUC、SJIS 和 UTF-8。

`jcode` 库可以不使用正则表达式，利用开放类的功能，使得字符串的方法可以处理多字节文字。图 6-17 给出的例子是加载 `jcode` 库之

后，像 `tr` 这样的字符串操作方法能够正确处理多字节文字。

```
#!/usr/bin/ruby -Ke
s = "abcde"
p s.tr("a-z", "A - Z") # => " Z 桥\332"
require "jcode"
p s.tr("a-z", "A - Z") # => "A B C D E "
```

图 6-17 用 `jcode` 库对 `tr` 方法进行功能扩展

`tr` 方法可以替换文字。图 6-17 是将半角的字母替换为全角大写的字母。但是，在加载 `jcode` 库之前的缺省状态下，`tr` 方法不能正确地处理多字节文字，文字发生了乱码（Z 桥...）。

加载 `jcode` 库之后，`tr` 方法“理解了”多字节文字，所以正确进行了替换处理，全部替换成了全角大写的字母。

`jcode` 替换表 6-4 中的 `String` 类的方法，实现对多字节文字的处理。表 6-4 中“破坏性的方法”是指将字符串本身进行替换的方法。例如，`s.squeeze` 把字符串 `s` 中连续的文字压缩成 1 个文字来返回一个新的字符串。而“!”的方法则是替换字符串本身内容，`s.squeeze!` 改变变量 `s` 本身所指向的字符串。

表6-4 `jcode`库中替换的`String` 类的方法

方 法 名	功 能
<code>chop</code>	删除字符串末尾的1个文字
<code>chop!</code>	<code>chop</code> 的破坏性方法
<code>delete</code>	删除指定的文字
<code>delete!</code>	<code>delete</code> 的破坏性方法
<code>each_char</code>	对各文字单位循环（追加）
<code>jcount</code>	指定文字的次数（追加）
<code>jlength</code>	文字数（追加）
<code>jsize</code>	<code>jlength</code> 的别名（追加）
<code>reverse</code>	文字的反转（1.9版）
<code>reverse!</code>	<code>reverse</code> 的破坏性方法（1.9版）

squeeze	连续文字的压缩
squeeze!	squeeze 的破坏性方法
succ	下一个字符串
succ!	succ 的破坏性方法
tr	文字替换
tr!	tr 的破坏性方法
tr_s	替换文字同时做连续文字压缩
tr_s!	tr_s 的破坏性方法

在多处引用同一个对象的情况下，使用破坏性方法会导致意想不到的字符串改写，所以有一定危险性。但是，因为不需要生成新的对象，执行性能会有所改善。

下面来看看 **jcode** 是如何实现对字符串单位进行操作的。图 6-18 是 **jcode** 库定义的 **String** 类的 **delete** 方法的代码。

```
class String
  def _regex_quote(str)
    str.gsub(/(\\\[\\]\-\\)|\\\.|([\\\[\\])|)/) do
      $1 || $2 || '\\' + $3
    end
  end
  DeletePatternCache = {}

  def delete!(del)
    return nil if del == ""
    self.gsub!(DeletePatternCache[del] ||= /[#{_regex_quote(del)}]+/, '')
  end

  def delete(del)
    (str = self.dup).delete!(del) or str
  end
end
```

图 6-18 delete 方法的实现

delete 使用正则表达式的 **gsub** 方法。需要注意两点。首先，要删除的字符串中可能存在有正则表达式中具有特殊含义的字符，所以用 **_regex_quote** 方法对这些特殊字符作转义处理；其次，为了降低每

次生成正则表达式的成本，在 `DeletePatternCache` 的表中保存了正则表达式。

原有方法在 `delete` 例子中虽然没有使用，但若想要在删除之后还能使用的话，可以用 `alias` 起个别名，把原来的方法保存起来。例如，在新定义的 `succ` 方法中，当字符串不包含多字节文字时，就使用原有的 `succ` 方法（参见图 6-19）。

```
alias original_succ! succ!
def succ!
  # end_regexp 是匹配字符串末尾的"文字"
  reg = end_regexp
  if $KCODE != 'NONE' && self =~ reg
    succ_table = SUCC[$KCODE[0,1].downcase]
    begin
      self[-1] += succ_table[self[-1]]
      self[-2] += 1 if self[-1] == 0
    end while self =~ reg
    self
  else
    original_succ!
  end
end

def succ
  str = self.dup
  str.succ! or str
end
```

图 6-19 succ 方法的实现

6.2.5 猴子补丁的几点问题

如上所述，开放类和猴子补丁让我们很方便地做各种操作，但是它们也有缺点。猴子补丁动态地对类进行变更，从而对程序整体产生了影响。特别是像 `mathn` 这样的库是非常危险的，因为它大幅度地改变了已有类的行为。程序代码本来以为整数相除会返回整数，可是如果在某一状态下包含了 `mathn`，很难想到它就是造成程序错误的原因。到现在为止都是正常执行的程序，只因为一个“`require`”而发生了错误。另外，在多个库都同时使用开放类对已有类进行变更时，如果相互之间发生矛盾则没有办法可以避免。

考虑到这些优缺点，想要正确使用开放类，安全地打猴子补丁，必须遵守以下几条规则。

基本上只是追加功能

对类追加新方法不会让已有的程序无法执行。使用开放类时，主要做不容易导致问题的功能追加会更保险。做功能追加时，如果发生名称重复时会造成麻烦，在选择追加的方法名时需要慎重。

进行功能变更时要慎重，尽可能小规模

像 `mathn` 那样有本质部分的功能变更可能导致预想不到的副作用。`jcode` 将所有的字符串处理都替换成以文字单位来进行，如果整体程序中有某处期望的是字节单位的处理，则会引起误操作。

利用开放类对已有的方法进行替换时，增加可选参数，或是只在特定的情况下进行变更，在一定程度上保持兼容性的变更会更保险。

小心相互作用

开放类，和继承、导入等其他处理方式相比独立性较低，相互之间功能容易受影响，所以一定要谨慎使用。追加的功能如果名称发生冲突，两边都在使用的话，没有解决这种矛盾的办法。

如果要使用多个利用了开放类的库，则必须慎重考虑这些库之间是否互相矛盾。

6.2.6 其他办法

猴子补丁能够不改变源代码进行动态修正，这种灵活性是显示动态语言柔软性和扩展性的好例子。可是，实现猴子补丁的 `Ruby` 开放类有时功能过强，可能会引起问题。

其他语言中用更易控制的形式也能实现猴子补丁。这里，介绍几种其他的办法。

C#

C#是微软的.NET 的中心语言，其热门表现目前还在不断改进。C#使用部分类和扩展方法来实现开放类和猴子补丁的功能。

部分类是指可以在多个场所进行分散定义的类。将类的定义分散在多个场所，从而实现了相当于开放类的功能。不过，和开放类相比较，C#有以下固有的限制：

1. 所有的部分类必须加上 **partial** 这样的专用语（不能在事后对任意的类进行扩展）；
2. 部分类之间不允许有矛盾（类不能够被覆盖）。

可以说，在开放类实现的各种功能中，部分类实现了可以把类定义分散开来这一特殊功能。

而扩展方法是指允许和调用普通方法一样调用的静态方法（参见图 6-20）。带 **this** 关键词的 **static** 方法就好像类中的方法一样被调用。

```
class StringExtensions
{
    public static string SwapCase(this string s)
    {
        .....
    }
}

class ExtensionMethodTest
{
    static void Main(string[] args)
    {
        string s = "Hello World!";
        Console.Write(s.SwapCase());
    }
}
```

图 6-20 C#的扩展方法

不过，扩展方法不能给已有的方法加钩子，或是进行替换，因为狭义的猴子补丁不能使用扩展方法。这也是仅仅实现了开放类一部分特有的性质。

CommonLisp

CommonLisp 提供了叫 CLOS（CommonLisp Object System）的面向对象功能，它实现了和其他面向对象语言风格不同的面向对象编程。Lisp 是本来就具备开放类功能的语言，加上它之后，CLOS 具备了“方法结合”的功能，可以实现方法的替换和钩子功能（参见图 6-21）。

```
(defclass example-class () ;; 没有父类
  ())                      ;; 没有实例变量

(defmethod do-something((param example-class))
  (print "do something!"))

(defmethod do-something :before((param example-class))
  (print "enter do-something"))

(defmethod do-something :after((param example-class))
  (print "exit do-something"))

(setq e (make-instance 'example-class))
(do-something e)
;;; 输出:
;;; "enter do-something"
;;; "do something!"
;;; "exit do-something"
```

图 6-21 使用 CommonLisp 的 CLOS 功能的代码示例

在方法的名字后面加上“:before”或“:after”成为了钩子。在这个例子中虽然没有出现，还有一个“:around”的指定，实现方法的替换。

前面说到 Lisp 的面向对象功能实现风格不同。看了图 6-21 的程序之后，对类和方法的定义及调用等哪些地方可以称为面向对象，很多人可能很困惑。

图 6-18 的程序调用方法用了下面的方式：

```
(do-something e)
```

相对于 Ruby 使用的方式：

```
e.do-something
```

在调用方法的印象上确实比较弱，好像只是在调用函数。

但事实上，**do-something** 这个函数（代表了多个方法的函数，可称为泛型函数）根据参数类的不同选择适当的方法。不管看起来给人的印象如何，从实际的行为来看，它和 **Ruby** 语句中把操作对象放在前面的方式是一样的。

这种写法接下来还有别的含义。和把操作对象前置的形式不同，函数形式不区分操作对象和其他参数，因此，**CommonLisp** 在函数有多个参数时，根据参数类的组合而确定不同的方法。像这样根据多个类的组合而调用方法称为多重方法（**multi-method**），它是 **CommonLisp** 面向对象编程的特点。

一般的面向对象语言，方法从属于类，选择方法名来调用。而 **CommonLisp** 的方法从属于名字（泛型函数），由参数的类（群）来选择。我们有趣地看到，同样是面向对象编程，不同的侧面就像是纵切和横切下去那样，呈现的面貌很不一样。

Lisp 的面向对象功能在较早阶段就和 **Smalltalk** 分离开，独立进行了发展进化。

面向方面

面向方面的目的是要分离出横向关联的共通侧面。面向对象语言把程序分割成对象单位，但世上并不是所有的共通侧面都能分割成对象单位。横向关联涉及多个对象的处理，共通侧面虽然只有一个，却断断续续地分散到多个类中。

比如，关于“记录日志”这个处理，想要记录日志的类有多个，那么在这个或那个类里都要有记录日志的处理。本来是想在一个地方集中处理所关心的事情，但因为程序构成已经被分割成类的形式，横跨了多个类的这个方面无法总结在一起。

像这样横跨了多个类的共同关注的事就是方面（**aspect**）。字典上 **aspect** 的解释有“外观、样子、局面、见解等”。按照关心的事来划分、记述并组合而成的程序称为面向方面编程。支持这种记述的语言称为面向方面的语言。

从可以直接记述这方面的意义来看，**Ruby** 不是面向方面语言。但是，**Ruby** 非常灵活，利用它的反射功能（操作程序本身的功能）可以实现面向方面编程的库，也就是 **AspectR**。**AspectR** 是 **Avi Bryant** 和 **Robert Feldt** 的作品。

使用 **AspectR** 的最初的例子如图 6-22 所示。

```
require 'aspectr'

class ExampleClass < Object
  def do_something
    p "do_something!"
  end
end

class Logger < AspectR::Aspect
  def log_enter(method, object, exitstatus, *args)
    p "enter #{method}"
  end

  def log_exit(method, object, exitstatus, *args)
    p "exit #{method}"
  end
end

Logger.new.wrap(ExampleClass, :log_enter, :log_exit,
  :do_something)

e = ExampleClass.new
e.do_something
# 输出
# "enter do_something"
# "do_something!"
# "exit do_something"
```

图 6-22 使用 **AspectR** 的记录输出

方面定义为 **AspectR::Aspect** 的子类。在这个类中定义作为钩子的方法。本例中定义了在执行方法之前调用的 **log_enter** 方法和执行方法之后调用的 **log_exit** 方法。这样的方法称为通知（**advice**）。在前面执行的称前置通知（**before advice**），后面执行的称后置通知（**after advice**）。**CommonLisp** 有环绕执行方法本身的环绕通知（**around advice**），但是 **AspectR** 没有。

调用 AspectR 的 `advice` 方法的参数如下：

第 1 参数	方法名（字符串）
第 2 参数	操作对象
第 3 参数	终止信息（返回值、异常）
第 4 参数之后	传递给原方法的参数

`before advice` 在方法执行前调用，没有终止信息。它的值总是 `nil`。

为了使用像这样定义的方面，必须把它和实际的类连接起来。于是先生成方面类的对象，调出 `wrap` 方法把类和方面连接起来。`wrap` 方法的参数如下。

第 1 参数	类
第 2 参数	<code>before advice</code> 名（或是 <code>nil</code> ）
第 3 参数	<code>after advice</code> 名（或是 <code>nil</code> ）
第 4 参数之后	钩子对象的方法名

图 6-18 的程序只挂了一种钩子，所以生成方面的对象之后马上就调用 `wrap` 方法。如果定义了多个钩子，或者为了能在之后卸下钩子，就需要把方面对象保存在某个变量中。图 6-15 的程序和图 6-22 的程序动作都是完全相同的。需要注意的是，`ExampleClass` 类的定义部分只记述了 `ExampleClass` 的处理本身，把记录日志这样所关心的事完全分离到 `Logger` 类里了。如果某一天不需要记录日志的时候，不管有多少方法曾经记录日志，只要把给 `ExampleClass` 类赋予记录日志功能的相关行删除掉，就可以把日志功能全部卸除了。如果把日志功能分散在各个方法中的话，卸除的时候就会很麻烦了。

6.2.7 Ruby on Rails和开放类

使用 Ruby on Rails 之后，经常会发现在 Rails 以外的 Ruby 程序中不怎么见得到的表现方式。比如下面的代码。

```
expire = 2.weeks.ago
```

此代码照一般的 Ruby 程序理解的话会是：对 2 这个整数对象调用 `weeks` 方法，对返回值再调用 `ago` 方法，把它的返回值赋给变量 `expire`。这么解释好像是没有疑问。但问题是，Ruby 标准的整数对象中没有定义叫 `weeks` 的方法。

实际上，`weeks` 是 Rails 的构成部分之一的 `ActiveSupport` 库提供的方法。`ActiveSupport` 利用 Ruby 的开放类功能，对 Ruby 标准提供的类大胆地追加了功能。`weeks` 方法是其中之一。

要点在于，`ActiveSupport` 库像打猴子补丁那样对整数类追加了 `weeks`、`ago` 这些方法。这种大胆性可以说正是 Ruby on Rails 的重要特点。

本人从以前就开始使用 Ruby，这种大胆让我感觉有点晕。而对于从 Rails 开始使用 Ruby 的人来说，反过来，没有这类的方法可能就觉得不好用吧。像前面提到的这些，标准库提供的功能对于程序语言给人的印象会产生很大影响。比如利用了 `ActiveSupport` 从 Rails 入门到 Ruby 的人和直接从 Ruby 入门的人，尽管是对同样的 Ruby，各人的印象也会有很大不同。

下面，概括一下 `ActiveSupport` 追加的功能以及它们的使用方法。

6.2.8 ActiveSupport带来的扩展

`ActiveSupport` 库对于 Ruby 语言提供了各种各样的扩展。`ActiveSupport` 库整体具有相当的规模，先来对追加的功能按照目的不同作个大致的分类。

首先，看看时间和时刻的操作。`ActiveSupport` 库对于 `Numeric`、`Time` 等类追加了时间和时刻的操作，比如前面说的“`2.weeks.ago`”。

具体来说，定义了表 6-5 中的那些方法³。

3 表 6-5 中的 `seconds` 等复数名词也定义了单数名词的别名。

表6-5 ActiveSupport库中关于时间的方法

Numeric 类	
方法名	功 能
seconds	秒
minutes	分 (×60)
hours	时 (×3600)
days	天 (×86400)
weeks	周 (×604800)
fortnights	2周
months	月
Numeric 类	
方法名	功 能
years	年
ago(t=Time.now)	从现在开始过去的n秒
until(t=Time.now)	ago 的别名
since(t=Time.now)	从现在起未来n秒
from_now(t=Time.now)	since 的别名
Time 类	
方法名	功 能
days_in_month	月的天数
seconds_since_midnight	00:00开始的秒数
change(options)	新时刻
advance(options)	未来的时间
ago(sec)	过去的时间
since(sec)	未来的时间
months_ago(n)	n个月以前
months_since(n)	n个月以后
years_ago(n)	n年以前
years_since(n)	n年以后
last_year	去年
next_year	明年
last_month	上个月

next_month	下个月
beginning_of_week	周的开始
next_week	下周
midnight	当天的00:00
beginning_of_day	midnight 的别名
beginning_of_month	当月的开始
end_of_month	当月的结束
beginning_of_quarter	季度的开始
beginning_of_year	年的开始
yesterday	昨天
tomorrow	明天

那么，整数类的 **weeks** 方法返回了什么样的值呢？我们来看一看。

```
2.seconds      # => 2
5.hours        # => 18000
1.month        # => 2592000
1.year         # => 31557600
```

看起来是返回了相应时间的秒数。1 个月是按 30 天，1 年是按 365.25 天计算的。

Time 类的 **change** 方法和 **advance** 方法的参数比较复杂，在这里解释一下。**change** 方法的参数是指定了年（**:year**）、月（**:month**）、日（**:mday**）、时（**:hour**）、分（**:min**）、秒（**:sec**）以及微秒（**:usec**）的哈希表。时刻的各要素通过名字来指定，比较方便。未指定的要素按本来的时刻来算。

例如，2009 年的今天，以下语句会获取和今天同月、同日、同时刻的 **Time** 对象。

```
Time.now.change(:year=>2009)
```

`advance` 方法也是同样接受可省略的哈希表参数，哈希表中可以指定“`:years`”、“`:months`”，“`:days`”，各自代表年、月、日的增量。想获取 1 年零 3 个月之后的时刻，写法如下：

```
Time.now.advance(:years=>1, :months=>3)
```

对于时刻的操作会频繁出现，能够像英语似的表达方式 `2.weeks.ago` 也许能让人高兴。虽然这么说，但对于整数类导入这样的方法感觉过于大胆，现在还未真正导入到 **Ruby** 中。

这样的“冒险”能在库的层次上推进，可以说是开放类的威力吧。

6.2.9 字节单位系列

计算机的世界里经常碰到千、兆、百兆等单位。日常生活中虽然已经熟知“千是 1000 倍”的意思，但对于百兆是多少倍可能有些人就不能马上反应过来。特别是，计算机世界以二进数为度量单位，千不是 1000 倍而是 1024 倍⁴ 的说法，听起来就更奇怪了。

4 IEC（国际电工委员会）在 1999 年规定，为了和 ISO 单位明确区分开，1024 倍称为 kibi，再 1024 倍称 mebi，再 1024 倍称 gibi。不过，此规定未获得推广。

ActiveSupport 能够减轻这样的麻烦，导入了一些方法（参见表 6-6）⁵ 可以容许下面的计算：

```
45.kilobytes + 2.6.megabytes
```

5 同样准备了 `kilobyte` 这样的单数别名。

表6-6 向**Numeric** 类追加的字节

单位系列方法	
方法名	处理内容
bytes	不变

kilobytes	1024倍 (2^{10} 倍)
megabytes	2^{20} 倍
gigabytes	2^{30} 倍
terabytes	2^{40} 倍
petabytes	2^{50} 倍
exabytes	2^{60} 倍

6.2.10 复数形和序数

日语里单数复数没有什么区别，但是英语里有明确的区分。因此，我们追加了专用的方法。比如，**pluralize** 方法可以让单数变成复数。

```
"box".pluralize      # => boxes  
"ox".pluralize       # => oxen  
"fish".pluralize     # => fish
```

也有方法用来操作多个单词连在一起的符号（参见图 6-23）。

```
"fish_burger".camelize  # => "FishBurger"  
"SoapDish".underscore  # => "soap_dish"
```

图 6-23 操作复数单词的方法举例

对于整数追加了序数方法：

```
1.ordinalize         # => "1st"  
3.ordinalize         # => "3rd"  
4.ordinalize         # => "4th"
```

像这样对名字进行操作的方法在英语圈可能会受欢迎。但 **Ruby** 是在日本开发的，对导入这种以英语为特定语言的语法操作的方法有抵触

感。不过像这种以库的形式对方法进行追加也许效果会更好。

6.2.11 大规模开发和Ruby

经常听到“Ruby 不适用于大规模开发”这样的说法。这当然不是毫无根据的说法，有它的道理。Ruby 做大规模开发可能会出现的问题主要有以下 3 个方面。

编译时不作类型检查

Ruby 不实际执行的话就检查不出类型的不一致。像 Java 那样在编译时就严格作类型检查一定会被查出来的错误，有可能在 Ruby 中就被漏掉了。因此有人认为，大规模程序的可靠性就下降了。

但是再细一想，程序的错误并不都是因为类型不匹配造成的。当然，类型不匹配是较容易发现的错误。Ruby 在执行时作类型检查，也就是说执行时会严格检查类型。大规模程序为了保证可靠性一定会有严格的测试程序。如果作了严格的测试，在编译时作类型检查的优点就不像所说的那么重要了。

没有包

Java 对于构成库的类和文件有独立的包，要想具备某种功能，必须明确地进行 `import` 操作。而 Ruby 是不具备这种功能的。所以，库定义的类和模块名是全局的，从任何地方都可以引用。因此，可以说名称重复的危险性很大。

不过，Ruby 有处理命名空间的类或模块。只要把库适当地组织好，发生名称问题的危险性也不见得会有那么高吧。

但在 Ruby 中，当互相独立开发的库凑巧定义了同名的类时，问题就没有那么容易解决了。在这种情况下，有必要对库的源代码进行修正。这样考虑的话，危险性不能说是零，而 Java 的包在这种情况下就能解决问题，所以可能觉得更好。

存在开放类

最后就是本章说明的开放类。已经讲述过，开放类有这样的缺点：各自独立的库发生互相矛盾的变更时，问题不能简单解决。这也可能在大规模开发时引发问题。

6.2.12 信赖性模型

但是，来看看上述 3 个方面会导致问题的大规模开发是什么样的情况吧。

编译时作类型检查能发挥重要作用，意味着不能充分执行单元测试。实际上，对于规模很大的程序整体作严格的测试，光测试本身就要花好多天，有可能在规定时间内都没办法完成。的确，像这一类的项目不推荐使用 Ruby。

因为不存在包而出问题或者因为开放类会出问题，这可能是因为项目的构成要素之间不太可能相互调整。如果把运用规则确定好，对各个子项目把模块分层次控制好的话，应该就没有问题。如果这都做不到的话，互相之间应该就是关系很坏吧。或者说各个子项目都是各自随便使用外部库，所以担心发生同名冲突。这样的项目也不适合用 Ruby。

Ruby 的这 3 个方面，依赖于 Ruby 和用户，或者项目的各成员间的“信赖”关系。也就是所谓的性善说。用户不会故意做坏事情，如果发生问题的话会互相帮助解决，这是 Ruby 所采取的姿态。

和它相对的是性恶说，即就不应该发生这些问题。不能说哪一边更糟糕，而是信赖性和灵活性的权衡罢了。

总而言之，像一部分人所说的那样，在某种类型的大规模开发中，Ruby 的性质会造成问题，或者说造成问题时解决起来不像其他语言那么容易，这种现象是现实中可能存在的。如果认为这些是问题的话，可能不使用 Ruby 会更好。但是，到现在为止我们看到的情况表明，会发生那种问题的大规模开发本来就绝不是好的开发状况。在我看来，首先要做的，当然是把项目的信赖关系改善到可以使用 Ruby 的程度。就算最后没使用 Ruby，这也是应该先做到的事情。

6.2.13 猴子补丁的未来

猴子补丁虽然有一定的危险性，但有弊也有利，它也提供了方便性、扩展性和灵活性。像这样危险和威力并存的情况，让人想起之前的 `goto` 语句。以前曾经是程序语言的标准控制结构的 `goto` 语句，随着结构化编程的发展，被更安全的控制结构取代了。与此相似，开放类和利用它的猴子补丁，将来也可能会被更安全的、由特定目的而特制的功能群而替代。

另外，虽然本节未作介绍，但为了避免因为开放类而改变整体状态的问题，对于 `selector namespace` 和 `class box` 的研究也快要实用化了。

像这样“驯服开放类”是将来 Ruby 研究的重大课题。Ruby 2.0 会对这一领域做某种程度的实质探索。

Ruby on Rails 的秘密

本章介绍的 Ruby on Rails 是在 2004 年问世的 Web 应用框架，因其生产效率高而成为近年来推进 Ruby 执行的原动力。

本人基本上没有使用 Rails 的经验，所以本章内容没有基于 Rails 本身，而是从 Ruby 的观点讲解了 Rails 使用的特征性技术。我不擅长 Web 类的编程。

Ruby on Rails 是 2004 年丹麦程序员 David Heinemeier Hansson 开发的。记不住他名字的人很多，经常称他为 DHH。他本来是 PHP 的程序员，虽然对 Ruby 感兴趣，但在实际的工作中并没有使用。DHH 为美国的 37signals 公司在自己家里上班，那个时候使用的是叫 Basecamp 的项目管理系统，因为它的复杂性而感到了 PHP 的局限。于是，他不顾周围反对而转移到 Ruby，结果成就了 Ruby on Rails。

经常作为 Rails 的特征而介绍的是 DRY（Don't Repeat Yourself）和 CoC（Convention over Configuration，约定胜于配置）。DRY 是避免反复（重复）从而提高生产效率的原则，CoC 是简化配置，重视约定，在一般情况下，这一原则排除明确的配置，从而使程序变得简洁。这些性质都是在应用平台上非常受欢迎的，所以在 Rails 之后也有很多 Web 应用框架采用这些原则。

可是，决定 Rails 生产效率的并不仅仅是 DRY 或是 CoC，而主要是利用了 Ruby 的元编程功能，几乎可以说达到了恶性滥用的程度。动态定义方法，用猴子补丁来替换类，简直是无所不能。可是，正是因为这样的彻底发挥才实现了 Rails 的强大功能，而且使之成为可能的正是 Ruby 的灵活性和兼容并蓄的宽大胸怀。所以，本书没有说明 Rails 本身的使用方法，而是着力于说明了之所以让 Rails 成为 Rails 的 Ruby 功能和编程技巧。当然，原因之一也是我本人不擅长 Web 应用开发。

关于 Rails 还有一个谜团，即为什么起名为“Ruby on Rails”呢？一般的软件名字都是 1 个单词或是“形容词 + 名词”，很少见到“A on B”这种形式。退一步讲，怎么说也应该是 Ruby 语言在下才对啊。以前就想见到 DHH 时直接问他，不过一直没有机会。上次见面时又完全忘记问了。

第 7 章 文字编码

7.1 文字编码的种类

计算机能够处理图像、动画以及各种应用程序固有的、多种多样的数据。但是从 CPU 的层次来看，计算机所处理的各种数据都是用比特 ON/OFF 所表现的二进制数字。

初期计算机主要用于炮弹的弹道计算等，所以专门针对数值计算而做的特殊设计也当然是最为理想的。但现在的计算机纯粹用于数值计算的已经越来越少了。像天气预报、结构计算或科学实验等的 HPC（高性能计算）都由超级计算机或是超级并行计算机来进行数值计算。但一般计算机所处理的数据，绝大部分都是以某种文字形式出现的文本数据。

7.1.1 早期的文字编码

为了让本来只能表示二进制数的计算机能够处理文字，就必须将文字变换为相应的数字。这种对应于文字的数值就称为文字编码。

由于历史的原因，文字编码遇到过各种各样的困难和课题。

初期的计算机是在英语国家发展起来的，计算机能处理的文字也是从英文字母开始的。英文字母只有 A 到 Z 这 26 个字母，没有元音变音（o 上加两点）、声调（e 上加声调）这些东西，处理起来比较方便，这也许与初期计算机的发展也有点关系吧。

表现英文的文字字符集的历史很悠久，其中有代表性的当数 1963~1964 年设计的 EBCDIC（Extended Binary Coded Decimal Interchange Code）和 1960 年开始制定的 ASCII（American Standard Code for Information Interchange）。

EBCDIC 由美国 IBM 公司定义，主要用于大型机及办公用计算机（据说有一部分现在还在用）。但现在 EBCDIC 已经不是主流，ASCII 以及受其影响的文字编码成为主流。

ASCII 码由 7 位二进制数构成，可以表现英文字母、数字和一些记号（\$、& 等），共 128 个字符。这带来的一个好处就是，对于通信单位的字节（8 位）来说，可以省出 1 位，用于附加错误检测码。

这在通信的可靠性还很低的时代，是一个很难能可贵的性质。7 位 ASCII 码能够将把整个字节的 8 位全部用于文字编码的 EBCDIC 码淘汰，也许就是因为这个原因。

表 7-1 及表 7-2 是 ASCII 编码表。比如，符号“*”由十六进制的 2A，十进制的 42 表示。Hello! 这个字符串由 72、101、108、108、111、33（十进制数）来表示。

表7-1 ASCII码表（十六进制）

	2	3	4	5	6	7
0		0	@	P	`	p
1	!	1	A	Q	a	q
2	”	2	B	R	b	r
3	#	3	C	S	c	s
4	\$	4	D	T	d	t
5	%	5	E	U	e	u

6	&	6	F	V	f	v
7	'	7	G	W	g	w
8	(8	H	X	h	x
9)	9	I	Y	i	y
A	*	:	J	Z	j	z
B	+	;	K	[k	{
C	,	<	L	\	l	
D	-	=	M]	m	}
E	.	>	N	^	n	~
F	/	?	O	_	o	DEL

表7-2 ASCII码表（十进制）

	30	40	50	60	70	80	90	100	110	120
0		(2	<	F	P	Z	d	n	x
1)	3	=	G	Q	[e	o	y
2		*	4	>	H	R	\	f	p	z
3	!	+	5	?	I	S]	g	q	{
4	”	,	6	@	J	T	^	h	r	
5	#	-	7	A	K	U	_	i	s	}
6	\$.	8	B	L	V	`	j	t	~
7	%	/	9	C	M	W	a	k	u	DEL
8	&	0	:	D	N	X	b	l	v	
9	'	1	;	E	O	Y	c	m	w	

7.1.2 纸带与文字表现

本书读者中，我想很多人都没现场看过 1966 年开始放映了不到一年的第一代《超人》。但也许有人在重播的电影中看见过操作员一边看着计算机吐出的带孔纸带，一边念着“东京湾出现怪兽”那种情景。

实际上纸带上每排有 8 个孔，用于表示一个字节（参见图 7-1）。每排的图案代表一个字符。操作员记住了代表每个字符的穿孔图案，所以也就能读出纸带上穿孔图案所代表的文字内容。不要说终端屏幕，

就连打印机都还比较稀有的时候，纸带就已经是很了不起的输出设备了。

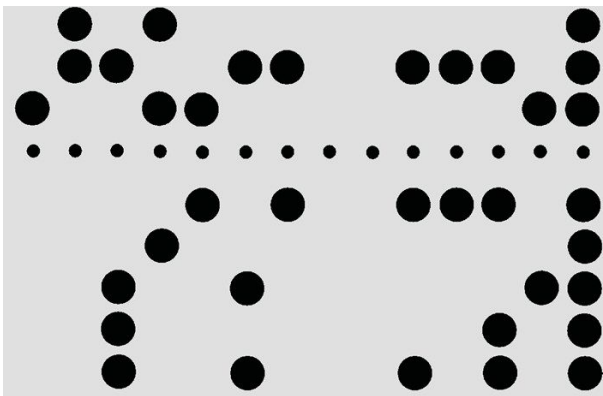


图 7-1 纸带扩大图。一行表示一个字节。中间一排小孔不是用于数据，而是用于运转纸带。最右边一行表示 DEL

而且，纸带还可以通过纸带读入器再度读入计算机，所以纸带也作为存储器使用。当时的程序大多是通过读纸带来执行的。一旦纸带上的程序有了 bug，只能用剪刀将纸带咔嚓咔嚓剪掉，然后再接上修正的方法，成为恰如字面意义上的补丁¹。bug（虫）这个词，也是因为计算机的继电器中夹了虫子所引起的事而得名。现在听起来像个笑话的事，在当时却是很平常的。

¹ patch 的第一个意思是打补丁。

还有一个老故事。ASCII 码中，删除字符串所用的 DEL 也分配了一个编码 127（十六进制的 0x7F）。这是为什么呢？0x7F 的 7 位全部都是 ON，用 DEL 覆盖既有字符时，既有字符就全部消失了。如果不知道纸带的故事，这也是很难想象的。

7.1.3 文字是什么

在学习文字编码的细节之前，先讲解一下关于文字编码的专用语（参见表 7-3）。

表7-3 与文字编码相关的专用语

用 语	意 义
文字	以视觉方式表现语言体系所用的符号

glyph	个别文字的字形
文字集	分配了文字编码的文字的集合
文字编码	分配给每一个文字的编码
文字编码方式	在计算机上表现文字编码的方式

首先是文字，这相当于人使用的一个个文字，但是事情并没有那么简单。比如日文假名的が字，将它看做是带浊点的单个平假名，还是看做平假名か（一个字符）再加上浊点（一个字符）的两个字符，要视具体情况而定。顺便说一下，个别文字的字形称为 glyph²。

2 汉字数据的“二”和片假名的“ニ”字形很相似，却是不同的字。

我不知道世上到底有多少文字，但要算上古代的文字，再加上根据需要随意创造的记号、图画和符号，这个数字事实上是无限的，同时处理所有文字是不可能的，所以有必要事先规定使用哪些文字。这些文字的集合就称为字符集（Character set）。

比如，刚才讲解的 ASCII 字符集，就是由英文的大写字母、小写字母、数字及特定记号所组成的字符集。除了 ASCII 字符集以外，还有欧洲语言用的 ISO8859，日语用的 JIS X 0208，以表现多语言为目的的 Unicode 等字符集。

字符集中，每个字符都分配一个编码，这称为字符编码。

计算机上仅仅用整数值来表示文字编码的方式称为文字编码方式（Character Encoding Scheme, CES）。一个字符集对应多种编码方式并不稀奇。比如 JIS X 0208 中的编码方式就有 ISO-2022-JP, Shift_JIS, EUC-JP 等几种。这些编码方式各有优缺点，使用状况也不一样。同样，对于 Unicode，也有 UTF-8、UTF-16BE、UTF-16LE、UTF-32BE 和 UTF-32LE 等编码方式。

这里所举的各种字符集和文字编码方式，后面还要个别说明。

虽然严格来讲，文字编码是指分配给文字的数值，但一般的对话中，使用文字编码这个词的时候，有时也包含字符集或文字编码方式等意思。在本章的标题文字编码中也隐含了这种意思。

7.1.4 走向英语以外的语言（欧洲篇）

为了表现英语以外的欧洲语言，26个文字通常不够。于是就使用ASCII中没有使用的第8位来表现文字。第8位一用，就可以再表现128个文字。虽然错误检测码不能再用了，但由于信息传送的可靠性提高了，好像也没什么问题。

即便同是欧洲语言，各语种中必须要加的字符也有所不同，于是就为每个文化圈定义了不同的文字集，文字集之间切换使用，这就是ISO 8859。现在，ISO 8859 中定义了16种文字集，列于表7-4中。

表7-4 ISO 8859的构造

标 准	分 部 名	内容（使用所定义文字的语言）
ISO8859-1	Latin-1	英语、法语、德语等
ISO8859-2	Latin-2	波兰语、捷克语、匈牙利语等
ISO8859-3	Latin-3	土耳其语、马耳他语、世界语等
ISO8859-4	Latin-4	爱沙尼亚语、拉脱维亚语、立陶宛语等
ISO8859-5	Latin/Cyrillic	俄语、保加利亚语、塞尔维亚语等
ISO8859-6	Latin/Arabic	阿拉伯语
ISO8859-7	Latin/Greek	希腊语
ISO8859-8	Latin/Hebrew	希伯来语
ISO8859-9	Latin-5	冰岛语、库尔德语、土耳其语
ISO8859-10	Latin-6（Latin-4重新定义）	爱沙尼亚语、拉脱维亚语、立陶宛语等
ISO8859-11	Latin/Thai	泰国语
ISO8859-12	Latin/Devanagari	印度各语言（1997年中断定义，成为废码）
ISO8859-13	Latin-7 Baltic Rim	Latin-4与Latin-6的追加文字
ISO8859-14	Latin-8 Celtic	苏格兰的凯尔特语、布尔顿语
ISO8859-15	Latin-9（改订Latin-1）	欧元货币符、法语辅助符等
ISO8859-16	Latin-10	阿尔巴尼亚语、意大利语、罗马尼亚语等

ISO 8859 所覆盖的语言，不管哪一种，其字符数都在256个以下，只要将文字编码以字节为单位排列起来就可以表现了。所以这些语言的字符编码方式跟ASCII一样是字节列。

这样欧洲圈的问题就大体解决了。但有些情况下，想同时使用德语（第一部分）和俄语（第五部分）等多种文字集。这样，就需要有在同一篇文章中切换不同文字集的机制。

定义这种机制的，就是 ISO2022。ISO2022 是一个规模庞大而复杂的规范，这里不再详述。基本上是以 ESC 文字（0x1b）为开头的 ESC 串来进行文字集之间的切换。正如后面所述，ISO2022 的机制也用在了日语等亚洲语系的文字编码中。

7.1.5 英语以外的语言（亚洲篇）

计算机刚开始在日本使用的时候，不能处理汉字及假名。所以，日语只能用罗马字来表示，这太不方便了，就用与 ISO 8859 同样的方法定义了包含片假名的文字集，这就是 1969 年定义的 JIS X 0201（参见表 7-5）³。现在还偶尔能见着 JIS X 0201 在使用，也就是半角片假名。

表7-5 JIS X 0201 编码表

	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0		0	@	P	`	p				-	タ	ミ		
1	!	1	A	Q	a	q			。	ア	チ	ム		
2	”	2	B	R	b	r			「	イ	ツ	メ		
3	#	3	C	S	c	s			」	ウ	テ	モ		
4	\$	4	D	T	d	t			`	エ	ト	ヤ		
5	%	5	E	U	e	u			・	オ	ナ	ユ		
6	&	6	F	V	f	v			ヲ	カ	ニ	ヨ		
7	'	7	G	W	g	w			ア	キ	ヌ	ラ		
8	(8	H	X	h	x			イ	ク	ネ	リ		
9)	9	I	Y	i	y			ウ	ケ	ノ	ル		
A	*	:	J	Z	j	z			エ	コ	ハ	レ		
B	+	;	K	[k	{			オ	サ	ヒ	ロ		
C	,		L	¥	l				ヤ	シ	フ	ワ		
D	-	=	M]	m	}			ユ	ス	ヘ	ソ		
E	.	>	N	^	n	~			ヨ	セ	ホ	°		
F	/	?	O	_	o	DEL			ツ	リ	マ	。		

3 JIS X 0201 是 ASCII 的扩展字符集，但历史上曾经使用过 EBCDIC 中追加平假名而形成的 EBCDIK（日立制作所）。

JIS X 0201 的 7 位部分与 ASCII 相当⁴，8 位都用来表示假名。

4 但是，JIS X 0201 中以 ¥（日元符）来表示 0x5C（ASCII 中是 \），以 ¯（上划线）来表示 0x7E（ASCII 中是 ~）。

但我们平常使用的并不只是片假名。于是 1978 年又制定了包含我们平常使用的平假名、片假名以及汉字的文字集 JIS X 0278。当初以 JIS C 6226 开始标准化，从 1990 年的修正版开始称为 JIS X 0208。

包括我们使用的日语在内的亚洲语系，大家都知道，与欧洲语系比起来，文字种类要多得多。JIS X 0208 有 6879 个字符（其中汉字 6355 个）。有这么多字，一个字节就表现不了。JIS X 0208（估计是）世界上第一个用多字节表现一个文字的字符集。

JIS X 0208 用 16 位（2 个字节）来表现文字编码，最初的 8 位称为区，后面 8 位称为点。各个区和点分别对应 ASCII 码中可以表示⁵的文字，所以 JIS X 0208 有 94 个区，每个区有 94 个点。每个文字用 n 区 m 点这种形式的坐标来指定。比如，平假名的 あ 为 4 区 2 点，汉字 松 为 30 区 30 点。

5 ASCII 中可以表示的文字是表 7-2 中所示的十进制数 33 对应的“!”到 126 对应的“~”，共 94 个字符。

既然一个字不能用一个字节来表示，就必须用某种形式的复数字节来定义，这就是文字编码方式。JIS X 0208 的文字编码方式分为 3 大类，即 Shift_JIS 和 EUC-JP 和 ISO-2022-JP（JIS 码，Junet 码）。

Shift_JIS

Shift_JIS 虽然以两个字节来表示 JIS X 0208 的字符，为了利用本来已经存在的半角假名，就避开了半角假名的空间，也就是错开了这部分空间。错开这个词在英文中是 Shift，Shift_JIS 因此得名。这种编码方式在微软系列操作系统中长期使用，所以通称 MS 汉字码⁶。

6 微软曾经将 Shift_JIS 称为 CP932。1983 年，日本 IBM 和 NEC 分别独自扩展了 CP932，1993 年，微软将“NEC 特殊字符”和“IBM 扩展字符”加入了 CP932。也有人将 CP932 称为“MS 汉字码”。

Shift_JIS 以两个字节来表示 JIS X 0208 的字符。第一个字节使用与 JIS X0201 不重复的空间（0x81~0x9f, 0xe0~0xfc），第二个字节使用更广泛的空间（0x40~0x7e, 0x80~0xfc），包括重复的部分。Shift_JIS 中，平假名的あ以 0x82 0xa0 来表示，汉字的松以 0x8f 0xbc 来表示。

Shift_JIS 的最大优点是它最为普及。美国微软公司的操作系统自 MS-DOS 以来长期使用 Shift_JIS，美国苹果公司的 Mac 操作系统也是。结果，所谓的个人电脑上，Shift_JIS 可以说成为事实上的标准了。同时，含有过去曾大量用过的半角假名的数据还可以继续使用，这在当时也是很方便的。

可是这些优点逐渐变得不重要了。现在这两个公司的操作系统逐渐改用后面将要介绍的 Unicode 了。半角假名及数据一点不变直接利用，已经不再是必然的了。

Shift_JIS 也有缺点。为了避开半角假名空间，第一字节需要错开因而空间变窄，第二个字节里 ASCII 文字出现了。这样，就不能简单地判别每个独立的文字。很多情况下，非得从字符串的最开头开始扫描才能正确判别字符串的内容。

而且，第二个字节如果有“\”这种可能带有特殊意义的字符的时候，可能发生很麻烦的错误。

比如，在 Shift_JIS 中编写 [print“成绩表”] 这种程序的时候，表的第二个字节是相当于“\”的 0x5c，意味着字符串终端字符的双引号被转义（即双引号与“\”结合被认作别的字符），造成语法错误。

EUC-JP

EUC-JP（Extended UNIX Code for Japanese）是完全无视半角假名的空间，从而使与 JIS X 0208 的相互转换变得简单化的一种字符编码方式。EUC-JP 广泛用于 UNIX 系列操作系统中的日语处理。为了得到某个字的 EUC-JP 码，只要将 JIS X 0208 码中字节的第 8 个位置设为 1（ON）就行了。EUC-JP 虽没有为半角假名保存空间，但并不是不能

表示半角假名，而是在 JIS X 0201 的文字编码前面放置一个 0x8e 的方式来表示⁷。

⁷ 例如，ア（JIS X 0201 中是 B1）以 8EB1 表示。

与 Shift_JIS 比起来，构成 EUC-JP 码的多字节文字的每一个字节的第 8 位都是 ON，所以有容易识别、字符串处理简单、不易出错等优点。也不存在 0x5c 问题（\问题）。

另一方面，因为与包含半角假名的数据不兼容，想维护过去的数据时稍稍有些不便。还有，处理半角假名时，以两个字节来表示，Shift_JIS 所具有的文字显示宽度与字节长度相同这一优点也失去了。

ISO-2022-JP

ISO-2022-JP 以 ESC 字符串来切换，使用 ISO2022 框架⁸来表示 JIS X 0208 文字集，又称 JIS 码，或 JUNET 码。

⁸ 虽然不使用 ESC 字符串，但 EUC-JP 也是在 ISO2022 框架的范围内的。

从 ASCII 以外的文字集切换到 ASCII 文字集时，先送一个 3 字节的字符串 ESC(B。切换到 JIS X 0208 文字集时，送另外一个 3 字节的字符串 ESC\$B。所以，ISO-2022-JP 中要表示“まつもと matz”时，就成为图 7-2 所示的 18 个字节。

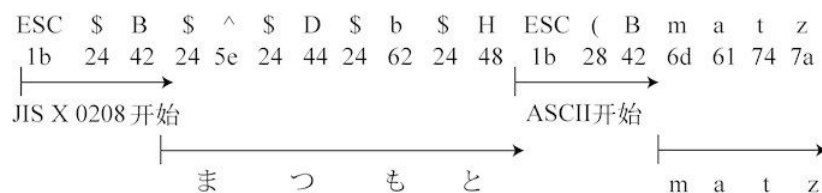


图 7-2 ISO-2022-JP 中所表示的“まつもと matz”

ISO-2022-JP 每遇到 ESC 字符串就改变状态。也就是说，即便是同样的一串字节，也会因为最近出现的 ESC 字符串的不同，而表现不同的文字集，具有不同的意义。这样的字符编码方式称为 **stateful**（带状态）。

反之，像 Shift_JIS、EUC-JP 那种不带状态的方式称为 **stateless**（不带状态）。带状态的字符编码方式在计算机内部处理时搞得很复杂⁹。

ISO-2022-JP 主要用于邮件及网络新闻等通信领域。

9 比如，从一个字符串的中间开始读出，如果不回溯到前面的 **ESC** 字符串，就不能够确定处理内容。

ISO-2022-JP 作为通信用字符编码方式而被采用的主要原因在于，以前，但也并不是很久以前，所使用的邮件系统中将通信内容的第 7 位给屏蔽了。像邮件、新闻那样，在到达最终地址之前，要经过多个服务器，不能因为一部分系统的问题而丢失信息，安全性尤为重要，所以第 8 位专用于校验。

实际程序中处理日语的时候，内部一般用 **Shift_JIS** 或 **EUC-JP** 来处理通信用的 **ISO-2022-JP** 码。

其他文字集

需要使用多字节的不只有日语。除日本以外的亚洲国家或地区也都各自定义了相应的文字集，像中国的 **GB 2312 (EUC-CN)**、韩国的 **KS X 1001 (EUC-KR)** 以及中国台湾的 **Big 5¹⁰** 等，都是有代表性的文字集（括号内是编码方式）。

10 **Big5**（大五码）并非公共标准，而是由 **Acer** 等 5 家电脑制造商自定义的文字编码。由于这个原因，文字集与文字编码方式没有分开。

7.1.6 Unicode的问世

各个国家都定义了自己使用的文字编码，结果世界上充满了各种文字编码。特别是在日本，光是日常使用的文字编码就有 3 种

（**Shift_JIS**、**EUC-JP** 和 **ISO-2022-JP**），它们还不能统一成一种。虽然没有日本那么严重，但如果不是使用 26 个拉丁字母的国家，或多或少都有类似问题。

如果软件只局限于一国一语，并没有太大问题，但往往不是那样。只使用英语的美国和英国的企业，将本国开发的软件扩展到海外及别的语言圈时，也会遇到问题。

为了解决这个问题，开始了两种动向。一个是 ISO 10646¹¹，以定义应该成为新时代的 ASCII 的文字编码体系为目的。另一个是 Unicode，在苹果、微软等软件公司主导（正确来讲是 Unicode consortium）下定义。虽然近期二者还会对立，但因为定义重复的文字编码是个浪费，以后会统一为一种。

11 国际标准化组织定义 ASCII 的是 ISO 646 标准，因此使用了 10646 这个名称。

本来对于 ISO 10646 的方案，软件处理上虽有点费事，但很通用。所以，最后决定统一为 Unicode 标准的时候，包括我在内，很多人都觉得可惜。但现在想起来，如果不能达成妥协，两种标准都残留下来的话，会比现在有更大的悲剧，统一了还是好。

Unicode 最初的目的是把世界上的文字都映射到 16 位空间中去。也就是说，当初预想，如果能用 65536 个文字¹²来涵盖世界各国的各种语言文字的话，我们就可以给每个文字分配一个唯一的编码。

12 $2^{16} = 65536$ （文字）。

正因如此，迄今为止以 8 位为单位的字符串，今后则以 16 位为单位来表示。这称为 UCS-2（2 byte Universal Character Set）。

7.1.7 统一编码成 16 位的汉字统合

如果将各国使用的字符编码一成不变地都收进来，65536 个字符怎么都不够。于是，对于字符个数最多的汉字，Unicode 中就将中国、日本和韩国所使用的意思相同的汉字分配了同一个编码。这称为 Han Unification（汉字统合，CJK 汉字统合）。其中 Han 是“汉”的中国读音。

结果，既存的文字编码与 Unicode 之间的变换不能通过计算来实现，只能通过内存效率低的变换表来实现。而且不能表现语言之间字体的不同。比如说，汉语的“骨”与日语的“骨”，上半部分内部的小方块的位置不一样。但在 Unicode 中被分配了同一个文字编码。

但 Han Unification 也不是只有缺点，像在文字排序和检索的时候，有超越语言、处理简单的优点。比如检索有关毛泽东的文章的时候，用

Unicode 的话，能跨越语言，不管是汉语还是日语的文章，都能随便检索。

以上的 16 位文字与 Han Unification 是初期的 Unicode 的显著特征。

7.1.8 Unicode 的两个问题

选择 16 位文字的 Unicode 有两大副作用，一个是字节顺序（byte order）的问题，一个是 NUL 文字问题。

计算机中表现 16 位整数的时候，关于字节顺序（字节按何种顺序排列）有两个流派。一个称为 little endian¹³，低位的 8 位先放。little endian 在美国英特尔公司的 x86 系列 CPU 中广泛使用。另一个称为 big endian，为美国 SUN 公司的 SPARC 等 CPU 所采用。

13 little endian 和 big endian 这两个词，来源于英国讽刺作家斯威夫特所著的《格列佛游记》（1726 年）。第一篇 Lilliput 国航行记中讲述，剥鸡蛋从尖的那一头开始剥的 Lilliput 国（little endian）与从圆的那一头开始剥的 Blefuscu 国（big endian）之间战争的故事。战争的原因在于剥鸡蛋的方法有分歧。

将 16 位字符串的 UCS-2 写入文件时，使用哪种 endian 是个重要问题。选了一方的话，在另一方的 CPU 上处理时要花更多成本。结果，Unicode 选了一种模棱两可的解决方案——哪种都可以。作为补偿，将 BOM 标志（byte order mark, BOM）放在文件头，以此判别文件中所含文本的字节顺序。

BOM 也分到了一个号码 0xfeff。这个号码的字节顺序反转以后得到的号码 0xffff 在 Unicode 中是空号，所以先读文件的前两个字节，如果是 0xff、0xfe 的话，这个文件的字节顺序就是 little endian，反之则是 big endian。如果先头两个字节既不是 0xff，也不是 0xfe，就假定字节顺序是 big endian。

如果字符串里混入了非 BOM 的标签，程序处理起来就变得复杂了。在先头以外的地方混入 BOM，或者稀里糊涂忘了在先头放 BOM 的情况总会有吧。最重要的是，字节顺序一旦错了，会引起可怕的文字乱码，所以字节顺序很重要。

除了字节顺序以外，16 位字符串引起的问题，还有一个叫 NUL 文字问题。

传统 C 语言处理的字符串，一般有一个终端文字 NUL（‘\0’）。但是作为 16 位文字的字符串，中途会出现 NUL 文字。所以，C 语言中处理字符串的传统函数（strcpy、strcmp）不能用于 16 位文字的字符串。

像 Java 那样的语言，一开始就是以 16 位文字为前提而设计的，所以没什么问题。但以 C 语言处理 16 位文字的时候，需要全新的 API。

7.1.9 Unicode 的文字集

单说 Unicode 的时候，往往是指作为文字集的 Unicode。Unicode 每次更新版本的时候，都要追加文字，现在几乎囊括了世界上存在的所有文字¹⁴。2006 年 7 月制定的 Unicode 5.0，实际上包含了 99024 个文字。

14 公元前 1400 年左右，希腊使用的“线文字 B”等历史上的文字也放进去了。

现在比 Unicode 更大的文字集只存在两个。一个是大约 16 万字的“今昔文字镜”，一个是 17 万字的“TRON code”¹⁵。从实用的角度看，“有了 Unicode 就能表现世界上的文字”这种说法也并非言过其实。

15 “今昔文字镜”（<http://www.mojikyo.com/>）以应用程序的形式提供，包含重复文字。“TRONcode”（<http://www2.tron.org/troncode.html>）有意增加了一些重复文字。

Unicode 有一个非常好的优点，那就是它不光收集了世界上的文字，而且同时提供了包含文字的名称、由来、意思以及文字类别等信息的数据库。这样，即便是读不出来的文字也能够处理。

但它并不是只有优点，有 3 个问题产生。第 1 个难点，因为不断增加文字，文字数已经突破了 16 位数的界限 65535。单纯的 16 位编码方式已经不能表示这么多字，显露出当初 Unicode 构想的不合理之处。

现在，Unicode 放弃了 16 位方式，而用 21 位来表示一个文字。现在 Unicode 的有效编码范围是 0~0x10ffff，能够表示 111 万 4111 个文字。不管有多少字，这肯定够用了。

7.1.10 文字表示的不确定性

Unicode 的另一个难点是为了表示某个文字，可以有多种方法。比如说，日文假名が（读作 GA），既可以看做是“304C-平假名字母 GA”，又可以看做是“304B-平假名字母 KA”与“3099-假名浊点符”的组合字符串。

幸运的是，Unicode 中定义了称为正规化（normalization）的手续。虽说有点麻烦，但可以将文字往尽可能紧凑的方向，或者往尽可能分散的方向统一。

第 3 个难点，这不能怪 Unicode，而是因为历史原因。但以 Unicode 处理日语的时候，是一个不可避免的问题。

ASCII 中 0x5C 被分配给了“\”（反斜杠），但在 JIS X 0201 及 Shift_JIS 被分给了[¥]（日元符）。在 Shift_JIS 的全盛期，用户需要用反斜杠时，用同一内码的日元符就可以了，不需要明确区别。

比如国外用\n（反斜杠+n）来表示换行符，日本用¥n（日元符+n）来表示，这些都无大碍。

但是，Unicode 中定义了反斜杠和日元符两个字符。这样在把旧文档转换到 Unicode 的时候，就无法自动判断，像旅行费¥50 000 的时候该用日元符，而 `print "hello\n"` 的时候要用反斜杠了。

另外还有多个字符也有类似的问题。

7.1.11 Unicode 的字符编码方式

为了表示 Unicode 文字集，有多种编码方式。主要有 3 种：UTF-8、UTF-16 和 UTF-32。根据字节顺序，UTF-16 和 UTF-32 又各自可以分为 2 类，分别是 UTF-16BE（big endian）、UTF-16LE（little endian）、UTF-32BE（big endian）和 UTF-32LE（little endian）。

UTF-8

UTF-8 为可变长的，与 ASCII 具有兼容性的编码方式。也就是说，将 ASCII 字符串当做是 UTF-8 字符串来处理也不会有问题。说得形象一

点，世上的 ASCII 码文件摞起来比山还高，这是一个很方便的性质。

另一个特征是以 UTF-8 编码的字符串，不含 NUL 文字。可以作为通常的 C 字符串来处理。从利用过去资产的角度来说，这也是个很便利的性质。

UTF-8 以一定式样的字节组合来表示 Unicode 中的 21 位文字（参见表 7-6），这样，操作 UTF-8 字符串时，可以利用以下便利性。

表7-6 UTF-8的字节组合式样及性质

字节组合式样	值的范围
0xxxxxxx	00-7f
110xxxxx 10xxxxxx	80-07FF
1110xxxx 10xxxxxx 10xxxxxx	800-ffff
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	10000-1fffff
111110xx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	200000-3ffffff
1111110x 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx 10xxxxxx	4000000-7ffffff

- 第一字节第 8 位（最上位）不是 ON 的字节，表示 1 字节文字（ASCII 的空间）。
- 只要看第一字节位的式样就能知道接下来还有几个字节，数数有几位是 1 就行了。
- 多字节文字的各个字节，除了第一字节以外，都以 10 开始（第 8 位是 1，第 7 位是 0）。即便不从字符串的先头开始扫描，也能知道构成这个文字的第一个字节在哪里。
- 采用以字节为单位的编码方式，没有字节顺序问题。

以上性质，对于内部程序处理字符串非常方便。另外，又没有字节顺序问题，在外部处理时也很有用。

但是，UTF-8 也有缺点。

- 消费过多的内存。如果仅限于 ASCII 空间，则与 ASCII 相同。但几乎所有的汉字，都要占用 3 个字节。
- 构成文字的字节数是可变的，因而在随机访问字符串中的任意文字时，代价与字符串的长度成比例。

但随着计算机内存的容量和性能的提高，到了现在，无视这些缺点也无所谓了。比如，即使 UTF-8 占用的内存是 Shift_JIS 的 1.3 倍，以 GB 为单位的内存和以 TB 为单位的硬盘，现在都很容易得到了，这些问题已经不成为问题了。反倒比接下来讲述的 UTF-16 和 UTF-32 更有好处。还有，随机访问有代价问题，但从长年处理 Shift_JIS 和 EUC-JP 等可变长字符串积累的经验与传统也知道，它几乎不会引起任何致命性的速度低下的情况。

UTF-16

Unicode 中能以 16 位表示的空间就以 16 位为单位来表示，超过 16 位空间的就以称为代理对（surrogate pair）的两个 16 位码的组合来表示。以前，Unicode 只包含 16 位就能表示的那些字的时候，UTF-16 被称为 UCS-2。

但是，Unicode 文字集从 2.0 版开始超越了 16 位的界限，到现在缺点已经很突出了。

自从 UTF-16 成为可变长之后，不光失去了 UCS2 所具有的 16 位固定长的优良特性，还增加了字节顺序等以前并不存在的问题。所以，UTF-16 已经没什么优点了，个人认为，从今以后，没有必要再采用这种字符编码方式了。虽说这样，以前选择 16 位编码方式（当时是 UCS-2，现在是 UTF-16）的平台还存在，完全抛弃也不行。采用 UTF-16 的平台有 Windows（文件路径的编码方式）和 Java（字符串处理）等。

UTF-32

UTF-32 采用 4 字节固定长的编码方式。耗费的内存量，在 ASCII 空间是 4 倍，在汉字空间也有 1.3 倍，因为这些原因，所以没什么人气。虽然如此，因为固定长，可以随机访问，作为内码也有些有利的性质，但存在字节顺序的问题，不推荐作为外部编码。

UTF-32 用得不是很广泛，在 Linux 的 libc 所提供的通配符字符串中有用到。

7.2 程序中的文字处理

首先复习一下关于文字编码的几个专用语。先确认一下文字编码、文字集和文字编码方式这 3 个专用语的区别。时不时被称为文字编码的 EUC-JP、Shift_JIS 和 Unicode，其实是属于不同的分类。EUC-JP 与 Shift_JIS 属于文字编码方式，而 Unicode 则是文字集的名称。

7.2.1 文字编码有多个意思

计算机不能处理文字本身，就要让文字与编码对应。与文字相对应的编码就称为文字编码。

以下要介绍的包含文字集和文字编码方式在内的文字处理方式，有时总称为文字编码。像这样，广义的文字编码属于不正确的表现，在严密意义上讨论时我们不用，而是使用正确的文字集或文字编码方式等词汇。

7.2.2 只能处理文字集中包含的文字

前面已经提到，计算机让文字与编码对应起来进行处理。反过来说，只能处理分配了编码的文字。分配了编码的文字集合就称之为文字集。

比如说，被称为 ASCII 的文字集是英文字母、数字及一些记号等被分配了 127 以下编码的文字的集合。日本使用的文字集合有 ASCII、JIS X 0201（通称半角假名）、JIS X 0208 和 Unicode 等（参见表 7-7）。

表7-7 日本国内使用的主要文字集

名 称	文字种类	文 字 数
ASCII	英文字母、数字、记号	96字（不包括控制字符）
JIS X 0201	英文字母、数字、记号、半角假名	159字
JIS X 0208	汉字、各种记号	6879字
Unicode	计划涵盖全世界的文字	99024字（5.0版）

7.2.3 纷繁复杂的文字编码方式

文字集确定以后，将各个文字对应的编码按顺序排列起来，就可以表示由文字排列成的文本了。

如果文字集中所用文字编码的最大值也小于 255（例如 ASCII），各个编码只要一个字节就可以表示，这里就不多说了。但是，在更大的文字集中，必须要考虑内存使用效率和处理效率等因素，来决定计算机如何处理。处置方法，换言之，就是把文字编码列的表示方法称为文字编码方式（Character Encoding Scheme）。

同一个文字集有多种文字编码方式的情形并不罕见。比如日本国内广泛使用的文字集 JIS X 0208 就有 ISO-2022-JP、Shift_JIS 以及 EUC-JP 等几种文字编码方式，它们各有优缺点，使用状况也不同。

同样，文字集 Unicode 有 UTF-8、UTF-16BE（big endian），UTF-16LE（little endian）、UTF-32BE（big endian）以及 UTF-32LE（little endian）等几种文字编码方式。它们也被适当地分开使用。

7.2.4 影响力渐微的Shift_JIS与EUC-JP

Shift_JIS 是从 MS-DOS 开始渐渐使用的面向 JIS X 0208 的文字编码方式。为避开 MS-DOS 以前曾使用的 JIS X 0201（半角假名）空间，将要分配的编码适当错开，因而得名 Shift_JIS。微软系列操作系统和 Mac 操作系统（汉字 Talk）都曾用过，在个人电脑领域具有压倒性的占有率。但近年来，Windows 和 Mac 相继改用 Unicode，Shift_JIS 的重要性日渐下降。现在已不推荐使用。

EUC-JP 是为了表示一个字节表示不了的，像 JIS X 0208 那种多字节文字，而将美国 AT&T 公司定义的 EUC（Extended Unix Code）进行日语化而得到的版本。那是面向 UNIX 所设计的。EUC 还有面向韩国的版本 EUC-KR 和面向中国的版本 EUC-CN。

EUC-JP 用于 UNIX 系列操作系统中日语的处理，今后仍会广泛使用，不过逐渐会被 Unicode 所取代。

7.2.5 Unicode有多种字符编码方式

日本有 JIS X 0208，中国有 GB2312，各国都定义了表示本国语言的文字集。这样，开发各国语言都能共同处理的软件就很困难。为了能共同处理全世界各种文字，就制定了文字集 Unicode。

当初，Unicode 的目标是在 16 位（65 535 个文字）的范围内表示全世界的文字。当时想，有这么多字应该够了吧。实际上，世界上使用的文字比预想的要多，不能够囊括在 16 位的范围以内。最新的 Unicode 5.0 发展为包含了 99 024 个字的巨大文字集。

Unicode 的优点是，对于文字集里的文字，还定义了包含文字的性质和作用等详细信息的数据库。

另外，应注意的一点是，根据所谓汉字统一（Han unification）的处理，中日韩三国意义几乎相同的汉字被分配了同一个文字编码。结果，不能通过计算从既有的文字集变换为 Unicode，而是需要定义一个很大的对应表。

Unicode 的文字编码方式，从大的方面分为 3 种。下面，按 UTF-16、UTF-32、UTF-8 的顺序来说明文字编码方式。

UTF-16

Unicode 中能以 16 位表示的空间（Basic Multilingual Plane，BMP）就以 16 位为单位来表示，超过 16 位空间的就以称为代理对的两个 16 位码（32 位）的组合来表示。

Unicode 只定义了 BMP 领域文字的时候，被称为 UCS-2。UCS-2 是 Unicode 制定之初推荐的文字编码方式。UCS-2 的上位互换方式为 UTF-16，用于 Java 中文字的内部表示¹，以及 Windows 和 Mac 操作系统的文件名表示等方面。

1 为了表示在计算机内部处理的文本数据而采用的字符编码方式称为内部表示。当内部表示和外部表示不同的时候，在读入数据时要进行变换。内部表示并不要求兼容性，但希望处理效率尽可能要高。

但现在 Unicode 已经超越 16 位，UTF-16 也不再有什么优点，没有必要再采用这种字符编码方式了。

UTF-32

UTF-32 以 32 位整数来表示 Unicode。不再有代理对那种丑陋的机制，每个字由一个固定长的数来表示。随机访问字符串中的各个文字时效率很高。

但在 ASCII 中只要 8 位就能表示的英文字母或数字，在 UTF-32 中却要占用 32 位，因而存在内存使用率低，以及字节顺序等问题，它不适用于通信以及文件保存等文本数据的外部表示²，而更适合用于字符串处理的内部表示。

2 为了表示往文件中保存的数据以及通信中传输的数据，这种超越一个程序范围的文本数据而采用的字符编码方式称为外部表示。外部表示并不要求处理效率，但在数据的空间效率、既有文本数据的互换性、表示的确实性，以及不易发生乱码等方面有要求。

UTF-8

UTF-8 为可变长的，与 ASCII 具有兼容性的字符编码方式。将 ASCII 字符串当做 UTF-8 字符串来处理也不会有问题。

另外，UTF-8 具有以下特征。

- 比 UTF-16 和 UTF-32 内存使用率稍微高些。
- 没有字节顺序问题。
- 因为是可变长的，所以文字的随机访问性能低。

因为与 ASCII 具有兼容性，而且没有字节顺序问题，所以对于外部表示来说，UTF-8 是一种具有优良特性的字符编码方式。UTF-8 虽然也是可变长的，如果借鉴同样也是可变长的 Shift_JIS 和 EUC-JP 在文字处理中所积累的可变长文字处理方法，则在内部表示领域也能够充分利用。

7.2.6 为什么会发生乱码

文字编码在理想情况下只有一个文字集（比如 Unicode）和一种字符编码方式（比如 UTF-8），这样字符串的处理就会变得很简单。但现实中却有数不清的文字集和为数众多的字符编码方式。

正因如此，现实世界中屡屡发生“文字乱码”问题。所谓文字乱码，是指文本数据不能正确表示和处理的状态。文字乱码虽然有各种起因，但共同的结果都是不能正确表示。

接下来，按原因分类，看看文字乱码都是怎么发生的。

7.2.7 字符编码方式错误

文字乱码的最大原因是把程序的字符编码方式搞错了。把一种字符编码方式的数据按另一种字符编码方式来表示，结果就会完全不同。汉字和假名等不能正确表示。

图 7-3 是将 EUC-JP 文本想当然地当做 Shift_JIS 而读入的例子。EUC-JP 的全角部分正好与 Shift_JIS 的半角相当，被读作半角假名。

```
EUC-JP:  R u b y は      強
          力
字节组合: 52 75 62 79 a4 cf b6 af ce cf
Shift_JIS: R u b y 、マ カ ッ
ホ      マ
```

图 7-3 由字符编码方式错误导致文字乱

几乎所有情况下，文本数据都不附加文字编码方式的信息，所以容易引起错误。这里，很多应用软件在读入文本数据以后，大体会按以下方法来尝试自动检出。

- 检查只在某种字符编码方式下才会出现的字节排列，从而来确定编码方式。
- 如果不能确定，则按照文字的出现频率来推测。

这样并不能100%确定字符编码方式。字符编码方式一旦错了，数据就完全没有意义了，产生非常显眼的字符乱码。

关于这一点，在邮件及经由 **HTTP** 传送的文本数据中可以指定字符编码方式³，比单单只有文本要强多了。但这个指定只是选项，并不是任何时候都可以用。

3 邮件是指 **HTTP** 邮件，**HTTP** 都有 **content-type** 域，可以写入 **charset=ISO-8859-1** 之类的字符编码方式的信息。

XML 在这方面很优秀。**XML** 的规范中明确规定，如果不指定字符编码方式，则都视为 **UTF-8**，不会发生“不知道是什么编码方式”的问题。被称为 **XML** 之父的 **Tim Bray** 曾说：“这难道不是 **XML** 的最大优点吗？”

7.2.8 没有字体

为了在计算机画面上表示文字，还需要文字编码及文字编码方式以外的东西。

与某种文字编码相对应的文字应具备何种字形的信息由计算机掌握，根据需要，必须将字形在画面上表示出来。这样的字形数据称为字体。

计算机想要表示文字时，如果没有对应的字体信息，就不能表示。这种情况下会发生某种乱码。

一部分处理系统对于这种表示不出来的文字，以空心方块来表示，所以有时把因字体不存在而发生的乱码称作“豆腐”。虽说没有字体，但只有一小部分文字不能表示，这在字符乱码中属于受害较轻的。

7.2.9 变换为内部码时出错

与文字编码方式错误的乱码类似的，还有编码方式变换错误。

正如后面将要介绍的，以 **Java** 为首的几种语言使用一定的内部编码方式。程序从外部读入的文本数据，不管是何种编码方式，**EUC-JP** 也好，**Shift_JS** 也好，**UTF-8** 也好，在进行处理之前，都先给它变成所谓的内部编码方式（很多情况下是 **UTF-16** 等 **Unicode** 系列编码方式）。

这个变换有时会发生判断失误。这样的话，就会发生与文字编码方式错误类似的乱码。

在表示时发生的编码方式错误，仅仅是表示的问题。但在变换时发生的错误，随着数据的变更，可能会破坏了数据。这样，以后想要通过更改编码方式来表示文本就难了。

7.2.10 发生不完全变换

使用“文字集”或“字符编码方式”这类词时，人们往往觉得 JIS（日本工业标准）等标准中会有严格的定义。但很多情况下，实际并不是那样。

比如说，很多手机都能使用各种各样的图画文字（参见图 7-4）。这些文字使用了分配给 **Shift_JIS** 的外字（用户可以自己定义使用的文字）领域中的文字及字形。但是，如何分配它们，各个电话公司都不一样。

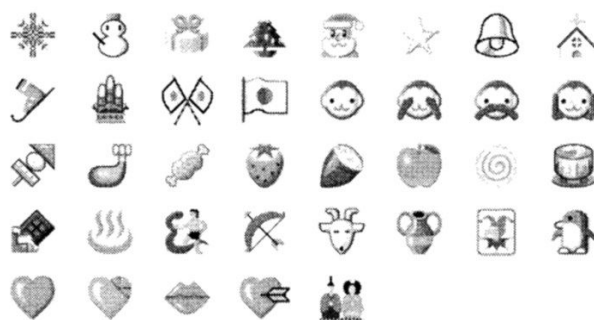


图 7-4 KDDI 的手机 (au 型号) 能够使用的彩色图画文字

从手机送来的文本数据，大体可以用 **Shif_JIS** 来处理。但不同的字符编码方式之间进行变换时，没办法知道手机的外字领域是如何使用的，不能指望变换正确。

现在就是这样一个问题，本来就是用户随便定义的外字，该怎么处理呢？如果事先知道了是哪个公司的手机送来的数据，向哪个公司的手机变换，就可以置换为类似的文字⁴。但现在，包含个人电脑在内，统一的处理还不可能。

4 2006 年 7 月，日本国内主要电话公司之间相互图画文字的变换服务开始了。名称虽然不一样，NTT docomo 叫绘文字变换功能，KDDI 和冲绳蜂窝电话叫绘文字互换服务，沃达丰（现在的 softbank mobile）叫邮件绘文字自动变换功能，但相互变换功能本身几乎一样。有很多符号，像心形符，在各公司间符号的形状和颜色几乎都能同等变换，但有些符号，像星座符，会变成形状和颜色完全不一样的东西，有些则变成了■（倒空）。

不只是手机，个人电脑也有依存于机种的文字（标准中没有规定，只在某些特定机种上可以利用）。特别是像 Shift_JIS 那种已经使用了很长时间的文字编码，为了提高便利性而偏离严格标准进行使用的现象很突出。

问题不仅在于机种依存的文字。为了能够正确变换，“原来的编码方式中的哪一个文字，对应于新编码方式中的哪一个文字”，这个问题需要毫不含糊地一一确定。但是，个人电脑中我们平常使用的文字，有不少就是在模糊地使用着。

本来在 ASCII（ISO 646）中，就可以根据各国情况的不同进行不同的文字映射。所以，JIS X 0201 中，ASCII 的反斜杠\所对应的编码（0x5c）被分配给了日元符号¥。波浪线~的编码（0x7e）则分配给了上划线¯。继承了 ASCII 的 Shift_JIS，原封不动地继承了这些文字。

正因如此，国外以反斜杠作为转意字符，而日本国内用日元符号作为转意字符这种怪事才会发生。

在作为 AT&T 国际定义的一部分规定的 EUC-JP 中，ASCII 部分不用 JIS 0201，而是用原始的 ASCII 码。严格来说，EUC-JP 不应用日元符号，而应用反斜杠。

分给这些文字的编码是共同的，现在只是表示上的问题。程序列表上的换行符，不管是\n，还是¥n，一看就能知道，并不是多么重要的问题。

即便以 JIS 标准为基础能进行机械变换的 Shift_JIS 与 EUC-JP 之间没发生问题，但 Unicode 要是掺和进来的话就不一样了。在将世界上很多的标准都吸收进来而诞生的 Unicode 中，\与¥，~与¯都有。这种时候，0x5c 该变成什么呢？

除了以上两个字符以外，Unicode 与其他文字集之间，不是一对一关系的文字还有几个。含有这种文字的数据进行变换时，会有丢失信息的危险。

7.2.11 文字集的不同

以 Shift_JIS 和 EUC-JP 为基础的文字集用相同字符编码方式进行变换时，除外字以外，不会出什么问题。但当文字集不同时，会发生在文字集中找不到应该变成的文字的情况。

向 Unicode 这种大文字集变换时，除部分模糊文字以外，大都可以如实进行变换。但是从 Unicode 这种大的文字集向其他文字集变换时，会遇到文字不存在的问题。

“这种文字不存在，错误！”“作为不能表示的文字，换成一个豆腐块（或倒空）。 ”⁵ 种种对策也存在，但不管怎么说，都是变换有问题。

⁵ 从活字印刷时代开始，有这种习惯，将不能处理的文字置换成■（倒空），因为将活字翻个个儿就印成那种字形。

想想也是，混杂着不能表示的文字，本来是混杂了这种文本数据的应用程序的问题，却要怪罪到变换程序头上来，真可笑。

7.2.12 字节顺序错误

像 UTF-16，UTF-32 这种基本数据单位大于 1 个字节的编码方式，存放数据时字节该以什么顺序摆放，有两大流派，一个是 big endian（BE），一个是 little endian（LE）。所以，同样是 UTF-16 格式，根据字节顺序不同就有两种，分别是 UTF-16BE 和 UTF-16LE（UTF-32 也一样）。这个搞错了，会引起很麻烦的问题（参见图 7-5）。

UTF-16LE:	g	a	z	z
字节组合:	7e 67 2c 67 4c 88 18 5f			
UTF-16BE:	最	愀	稀	稀

图 7-5 字节顺序错误导致文字乱码的例子

以上从文字编码方式错误到字节顺序错误，介绍了各种文字乱码。从影响很大的到微不足道的，有各种程度的乱码，原因也各不相同。我

们并非住在理想世界里，一时还会为文字乱码而烦恼。

但希望也是有的。世界上很多的文本数据都在慢慢变成以 Unicode 来表示。

长此以往，Unicode 广泛普及，Unicode 以外的文字集的文本数据都变成过去式，而且 UTF-8 变为主流（编码方式），（至少日常生活中）我们就有望从文字乱码中解放出来了。

7.2.13 从编程语言的角度处理文字

有了以上知识，来看看编程语言是如何处理文本数据，特别是文字编码方式的。

编程语言处理文本数据的方法，从大的方面分，有 UCS 方式和 CSI 方式两种。

7.2.14 以变换为前提的UCS方式

所谓 UCS（Universal Character Set，泛用字符集），是指程序中所处理的共同文字集（及字符编码方式）。输入输出时，编程语言将文本数据变成 UCS，内部对文本数据进行统一处理。UCS 被各种编程语言所利用。它具有以下优点。

- 原理简单，容易实现。
- 除变换外，处理成本低。
- 实际成果多。

但它也有缺点。

- 发生不必要的变换。
- 变换存在模糊部分。
- 有外字及机种依存文字的问题。

- UCS 中不包含的文字绝对不能处理。

内部用 UTF-16 时的不必要变换，可以举出一例：输入是 EUC-JP，输出也是 EUC-JP 的情形。如果按 EUC-JP 处理就不需要变换，但 UCS 中必须进行 EUC-JP → UTF-16 → EUC-JP 的变换。

7.2.15 原封不动处理的CSI方式

所谓 CSI (Character Set Independent, 字符集独立)，是指不对各种文字集（及编码方式）进行任何变换，原封不动进行处理。相对于 UCS 的内部只有一种编码方式的处理方法，CSI 中对各种编码方式原封不动地处理。

CSI 是优点多，自由度高的方式。

- 不发生不必要的变换。
- 不发生变换所带来的问题。
- 不易发生外字的问题（可以采取对策不令其发生）。
- 理论上不存在不能处理的文字。
- 根据需要，可以处理应用程序独立的文字集。

CSI 可以处理多种文字编码方式，所以是 UCS 的超集。实现 CSI 方式后，只要在输入输出时追加字符编码方式变换的编码，就成为 UCS 方式。

但与 UCS 相比，CSI 有以下缺点。

- 字符串的处理容易变得复杂化。
- 预计处理性能会变低。
- 实际成果少。

实际上，现在存在的多种编程语言中，采用 CSI 方式的几乎没有。Ruby 从 1.9 版开始提供 CSI 方式的字符串处理功能，成为少有的例外。

7.2.16 使用UTF-16 的Java

Java 采用 UCS 方式，内部的字符编码方式选用 UTF-16。

对 Java 而言不幸的是，UTF-16 本身的面貌已经从当初发生了变化。当 Java 确立字符串处理的基本部分的时候，Unicode 还是 Ver 1.0，只存在现在被称为 BMP 的部分。

Java 语言反映这一状况。Java 的字符型（char）是 16 位整数，字符串在内部以 16 位整数的数组来表示。结果，就产生了以下问题。

- 不属于 BMP 的文字（Java 中称为辅助文字）以两个字符来表示。
- 求字符串长度时，调用含有一个辅助文字的 String 的 length 方法，不是返回 1，而是返回 2。
- 像求字符串长度这种基本的字符串操作，也不能使用 String 类本身基于 char 的方法，而需要使用基于代码点（CodePoint，Java 中对文字编码的叫法）的新方法。

综上所述，如果想要正确处理辅助文字，不要使用旧版 Java 提供的 API，必须使用 Java5.0 以后版本提供的 API（参见图 7-6）。但是，现在登录进 Unicode 的文字一次又一次增加，只能处理 BMP 文字的程序已经不允许了。

```
//求字符串长度
void lengthTest(String s) {
    //基于char 求字符串长度
    System.out.printf("char len = %d\n", s.length());
    //基于codepoint 求字符串长度
    System.out.printf("codepoint len = %d\n",
                      s.codePointCount(0, s.length()));
}

//按文字循环
void iterTest(String s) {
    //基于char 对字符串循环
    for (int charIndex = 0; charIndex < s.length(); charIndex++){
        int c = s.charAt(charIndex);
        System.out.printf("%x\n", c);
    }
}
```

```

    }
    // 基于codepoint 对字符串循环
    for (int codeIndex = 0; codeIndex < s.length(); ){
        int c = s.codePointAt(codeIndex);
        System.out.printf("%x\n", c);
        codeIndex += Character.charCount(c);
    }
}

```

图 7-6 BMP 与辅助文字的处理代码不相同的 Java

制定 Java 时，Unicode 仅限于 16 位。Java 没选择可变长的 UTF-8，而选择了 UTF-16，因而产生了这样的悲剧。说是时机的恶作剧也罢，真是太可惜了。

虽说这样，看看图 7-6 的程序，基于 char 的程序和基于 CodePoint 的 API，也没那么复杂的差异。Java 的字符串处理的抽象化程度本来就不高，字符串处理本来就很复杂，所以 API 的影响也就很不明显。与擅长字符处理、各种文字处理都能以简洁的方式表现的 Ruby 那样的语言相比，Java 的文字处理怎么都显得冗长。所以，现在因采用 UTF-16 而使复杂度变得不那么明显，不知对 Java 是好还是坏。

7.2.17 使用UTF-8 的Perl

Perl 也使用 UCS 方式，内部编码方式采用 UTF-8。开始就采用可变长的 UTF-8 的 Perl，没有 Java 中由 UTF-16 所引起的问题。而且，与 Java 不同，字符串不是数组（几乎不具有数组的性质）这一点，也成为不必在意字符编码方式的理由之一。

Perl 的字符串中有“utf8 标志”。设定了标志的字符串，是以 UTF-8 编码的 Unicode 文本；没设定标志的字符串，以字节串来处理。Perl 原来也将文本数据及二进制数据同样用字符串处理，虽说是反映了历史情况，结果是 API 变得容易使用了。

只要读入的字符串没有明确指定，都被看做是 utf8 标志没设定。为了给文件的输入设定 utf8 标志，要么用 `use open` 语句对所有文件都设定，要么对每个文件在 `open()` 时设定，要么以二进制读进来之后再设定，哪种方法都可以（参见图 7-7）。

```

# 对文件全体指定
use open IN => ':euc-jp';           # 输入用EUC-JP
use open OUT => ':utf8';           # 输出用UTF-8
use open ':encoding (iso-8859-7)'; # 全部输入输出用iso-8859-7

# 对每个文件进行指定
open(I1, "<:utf8", "file");         # 输入用UTF-8
open(I2, "<file");                 # open
binmode(I2, ":utf");               # 以后指定

# 以字符串为单位变换
use Encode;
$string = Encode::decode("euc-jp", $data);
# $data 是二进制, $string 是 UTF-8 文本

```

图 7-7 Perl 中指定字符编码方式的 3 种方法

Perl 中，全部的字符串处理和正则表达式都是以 **utf8** 标志为前提而工作的，只要控制好输入输出，以 **UTF-8** 为基础进行的文本处理就不会发生任何问题。

至于 UCS 所特有的由文字变化带来的问题，Perl 是给 **Encode** 模块追加一个变换表来解决的。要说 **Encode** 模块是文字编码处理的中心也不过分。维护这个模块的是日本最有名的开源代码程序员小饲弹。有一种说法是，在 Perl 的源编码发布物中，有一半以上都是这个 **Encode** 模块变换表，真是很了不起。

7.2.18 用UTF-16的Python

世界上很有名的 Python，在日本的知名度从现在才开始增加。简单总结一下。

Python 也采用 UCS 方式，字符编码方式采用 **UTF-16**⁶。与按 **utf8** 标志的有无来区别字符串的 Perl 不同，Python 中表示二进制数据的字符串的类 **str** 与表示 **Unicode** 字符串的类 **unicode** 被明确区别开来。

⁶ Python 在编译时，根据选项可以将内部编码方式选为 **UTF-32**。实际上，类似 Fedora 和 Ubuntu 的 Linux 中，Python 指定 **UTF-32** 为内部编码方式进行编译。

二进制是 `str`，文本是 `unicode`，听起来有点怪怪的。但 Python 3.0 中，二进制是 `bytes`，文本是 `str`（Unicode 字符串）。但如果指定用 UTF-16 编译的话，代理对依然被算成两个文字。

7.2.19 采用CSI方式的Ruby 1.8

Ruby 从以前开始，因为可以不经变换直接处理文本数据而采用了 CSI 方式。

但是 1.8 版以前的 Ruby，有一个非常大的限制。能够处理的文字编码方式只有 `EUC-JP`、`Shift_JIS` 和 `UTF-8` 三种。考虑到世界上有那么多文字集，不可否认这会给人偷工减料的印象。不过 Unicode（UTF-8）覆盖了很广的范围，在实用上可以说没有问题。

Ruby 1.8 以前的文本处理中，每一个字符串对象不带字符编码方式的信息。Ruby 1.8 中正则表达式带有编码方式信息，以文字为单位处理时用正则表达式。字符串对象说到底还是按字节串操作（参见图 7-8）。

```
# 全部的文本处理都按EUC-JP 为基础
$KCODE='e'
#全部的文本处理都按UTF-8 为基础
$KCODE='u'

#UTF-8 对应的正则表达式（用u 选项指定）
re = /foo 汉字あいうbar/u
#以文字为单位分割string
chars = "abc あいう汉字".split(/u)
```

图 7-8 Ruby 1.8 的文本处理

通过在正则表达式的后面添加选项，可以指定对应的编码方式（参见表 7-8）。

表7-8 Ruby 1.8中正则表达式的字符编码方式选项

选 项	编码方式
u	UTF-8
e	EUC-JP

s	Shift_JIS
n	字节串

给全局变量\$KCODE 设定适当的值，可以给不带选项的正则表达式指定编码方式。

Ruby 的字符编码方式的对应与其他的语言很不相同。

- 不采用 UCS 方式。
- 字符串里不附加字符编码方式的信息。
- 字符串对象一直作为字节串而操作。

乍一看，觉得好像功能不够，实际用起来，居然还挺好。之所以这样风气一变，诞生了与其他语言不同的字符串处理方式，也许是因为这门语言生在长年受多字节字符串处理之苦的日本。

当然 Ruby 1.8 的字符串处理方式也不是没有缺点。它存在以下几个问题。

- 除了事先嵌入的 3 种字符编码方式以外，不能处理其他字符编码（最大的弱点）。
- 想以文字为单位进行字符串处理时，不经过正则表达式就不能直接处理。
- 在别的语言中能够做到的由名称及代码点（code point）进行的文字指定，Ruby 中不行（参见图 7-9）。

```
# 由名称指定αα
"\x{LATIN CAPITAL LETTER A}\x{GREEK SMALL LETTER ALPHA}"
# 由代码点指定αα弹
"\x{61}\x{3b1}\x{5F3E}"
```

图 7-9 由名称及代码点指定的例子

针对这些问题，2007 年 12 月公布的 Ruby1.9 改善了字符编码方式的处理。

7.2.20 强化了功能的Ruby 1.9

从 Ruby 1.9 开始，在沿用 1.8 版及以前的 CSI 方式的基础上，又进行了几项功能强化，其基本方针列举如下。

- 作为正则表达式引擎，将 1.8 版及以前的来自 Emacs 的正则表达式引擎更换为称作“鬼车”的引擎。
- 让个别字符串也能够附加编码方式信息。
- 根据附加信息的指令，可以按字符单位进行处理。
- 也能够追加用户定义的编码方式。
- 根据编码指示（coding pragma），可以指定编码方式。
- 嵌入编码方式变换方法。

用 Ruby 1.9 的功能编的程序大意示于图 7-10。

```
# -*- coding: utf-8 -*-  
# 文件开头的注释，指定本文件是UTF-8  
  
# 指定从文件中读入UTF-8  
open(path, "r:utf-8") do |f|  
  f.eachline do |line|  
    printf "f=%s\n", line.encoding # 显示 "f=utf-8"  
    print line.encode("sjis")      # 变换为 Shift_JIS 显示  
  end  
end
```

图 7-10 Ruby 1.9 支持编码方式的例子

7.2.21 是UCS还是CSI

Java、Perl 和 Python 等采用的 UCS 方式，从外部读入数据变换为 Unicode 后进行处理。这种方式可以对包含于 Unicode 的文字进行统一处理，程序结构变得简单。Unicode 被设计成世界上使用的很多文字集的上位集合，几乎在所有情况下都能做得很顺利。

另一方面，**Ruby** 所采用的 **CSI** 方式，文本数据尽可能不做变换地进行处理。正如文字乱码那一节说明的那样，由于历史上的原因，在文字集之间进行变换时，会出现各种各样的问题。**Ruby 1.8** 中，只能处理 **EUC-JP**、**Shift_JIS** 和 **UTF-8** 三种编码方式。但 **Ruby 1.9** 提供了更加彻底的 **CSI** 方式的字符串处理功能。

作为 **CSI** 方式的副作用，用户独自の字符编码方式的处理也可以定义了。所以，在不久的将来，像 **TRON** 码及今昔文字镜那种比 **Unicode** 更大的文字集，**Ruby** 说不定也能够处理。

* * *

编程处理文字的时候，残留着各种各样的历史事件和复杂的问题。文本不能正确显示的文字乱码就是其典型例子。为了不引起文字乱码，需要对文字集和文字编码方式有一个正确的理解。

Unicode

很久以前，在计算机能够处理的文字只有字母和几个符号的时代，世界曾是和平的。但文字就是文化。世界上的人类日常生活中所使用的不是只有字母，如果不让这些使用自己的文字，计算机未免也太傲慢了点儿。

于是，各个国家制定了表示本国文字的标准，各国的文字也都能表示了。但这样一来，各个国家都制定了不同的标准，搞得软件在每个国家都得修改，国际软件开发成本有些没谱。

为了解决这个问题，就诞生了 **Unicode**，但 **Unicode** 的诞生经历了太多困难。我认为主要有 3 个原因。

首先是政治的原因。当时，与制定文字集牵连很深的计算机（硬件）厂商，说实在话对新的文字集并不抱很大希望，所以，他们对 **Unicode** 基本上表现都很消极。另一方面，软件厂商为了降低软件国际化的成本，无论如何都需要一个能用的文字集。标准的确定怎么说都会带来利害冲突和政治问题。**Unicode** 虽然还有各种问题，但终究成为能用的文字集，不愧是个奇迹。真是太好了。

其次是文化的原因。文字是文化的反映，每个人都有各自的喜好。比如说，高桥这个名字中使用的高字，有的是“点横头”（⊥）下边为“口”字的“口高”，有的是下边像个梯子的“梯子高”。

不在意的人看起来，只是觉得字体不同。但在意的人看来，好像是完全不同的文字。仅 JIS 定义的文字集中，就有像“富”及“冨”这种区别很小的字。这种状况下，作出让所有国家的所有人都满意的决定是不可能的。对于 Unicode 的决定即便是那些心怀不满的人，也认为它大体是妥当的。

最后是技术的原因。当初，Unicode 假定世界上所有的文字都能囊括在 16 位以内，也就是 65536 个文字的范围。对于日常生活中只用 127 个文字的 ASCII 码就能满足的美国人来说，认为超越 6 万字的巨大空间已经足够了，某种意义上也是没办法的事。但是事实上，不要说 6 万字，Unicode 5.0 已经收录了 9 万多字，而且还没完没了。从这一点可以得到教训，“肯定没问题”这种迷信是很危险的。

作为日本的程序员，长期进行与文字编码相关的工作，得到的经验就是文字编码是棘手的问题。真心期待这些麻烦的问题会因 Unicode 而越来越少。问题是随着新文字集的出现，短期性的混乱还会越来越厉害。

第 8 章 正则表达式

8.1 正则表达式基础

正则表达式是处理字符串时所用的记述方法，在“从日志文件中提取像日期的东西”以及“从文章中找出似乎是拼写错误的东西”等情况下很有用。本节介绍正则表达式及其基本使用方法。

8.1.1 检索“像那样的东西”

在处理字符串的时候，“包含‘Ruby’的字符串”这种表达是比较简单的。但有些情况下，想用更模糊一点的规则来记述字符串的处理，比如“含有以 P 开始，以 l 结尾的单词”等。但计算机不懂人的语言，所以必须用计算机懂的语言来表达这种指示。这种记述规则就是正则表达式。在 Ruby 里，以斜杠 (/) 包起来的部分标为正则表达式，比如下面这种使用方法。

```
/Ruby/  # 字符串"Ruby"  
/P.*l/  # 以P 开始，以 l 结尾
```

正则表达式是描述字符串模式的一种微型语言，可以将其看做是嵌入到 Ruby 中的其他语言。因为它是个独立的语言，与宿主语言

(Ruby) 有着完全不同的语法，所以有时会成为混乱的根源。但若能好好掌握其语法，则没有什么比它更方便的了。

在 UNIX 系列操作系统上，很多语言和工具都能使用正则表达式。Ruby、Perl、AWK、sed、ed、vi、Emacs 以及 less 等都能用。正则表达式的语法本质上都一样，但各个工具都独立地实现了正则表达式，语法上多少有些差异。这里讲解 Ruby 里使用的正则表达式。

用正则表达式的模式匹配能做些什么呢？这里介绍其一部分功能。比如，如果使用模式匹配功能，以下的工作会变得很容易。

- 从日志文件中取出像日期的东西。
- 从 HTML 文件中取出像 URL 的东西。
- 典型的拼写错误检索。

“像那样的东西”这种表达，用模式匹配以外的方法很困难。而这正是模式匹配的拿手好戏。

8.1.2 正则表达式的语法

正则表达式由称为元 (meta) 字符的特殊功能字符和其他字符组成。元字符有点像编程语言中的控制结构。正则表达式中的元字符见表 8-1。

表8-1 正则表达式的元字符

元 字 符	含 义
.	与任意单个字符相匹配
[]	[a-z]与a到z之间任何一个相匹配
[^]	[^a-z]与a到z以外的字符相匹配
\w	构成单词的字符
\W	构成单词的字符以外
\s	空白文字。与[\t\n\r\f]相同
\S	非空白文字
\d	数字。与[0-9]相同
\D	非数字。与[^0-9]相同
*	重复0次以上
+	重复1次以上
?	重复0次或1次
{m, n}	重复m次到n次
*?	重复0次以上（懒惰匹配）
+?	重复1次以上（懒惰匹配）
??	重复0次或1次（懒惰匹配）
{m, n}?	重复m次到n次（懒惰匹配）
^	行首匹配
\$	行尾匹配
\A	字符串头
\Z	字符串尾（若含换行，匹配前一字符）
\z	字符串尾
\b	backspace(0x08)（[]内）
\b	词头或词尾（[]外）
\B	非词头亦非词尾
	选择
()	分组（有向后引用）
\1, \2	向后引用（对应第n个括弧）
(?:)	分组（无向后引用）
(?=)	用模式指定位置（无宽度）

(?!)	用否定模式指定位置（无宽度）
(?#)	注释

除元字符以外的字符与处理对象的字符本身相一致。比如，`/FOO/`与`FOO`的字符排列相一致。前面讲述的`/P.*l/`解释如下。

- **P** 是普通字符，与字母 **P** 本身相匹配。
- “.”与任意字符相匹配。
- “*”表示前面的模式“.”重复任意次。
- **l** 与字母 **l** 本身相匹配。

用人类语言表述以上内容，就是“以 **P** 开始，以 **l** 结尾”的字符串。也就是说，**Perl** 及 **Pascal** 都能与之匹配。“*”也包含 **0** 次，所以 **Pl** 也能与之匹配。**0** 次很容易被忘记，需要特别注意。

下面详细讲述正则表达式的语法。

普通字符

除表 8-1 中所示的元字符以外的普通字符，都与该字符自身相匹配。也就是说，**a** 与 **a** 自身相匹配。不含元字符的字符串的模式就是该字符串本身。比如与 **Ruby** 相匹配的模式就是 **Ruby**。

字符集合

用括号 (`[]`) 括起来的部分为字符集合。与括号内所含的每一个字符都匹配。比如，`[abcdef]` 能与小写字母 `abcdef` 中的任何一个相匹配。

字符集合中，用中划线 (`-`) 来指定范围。所以，`[abcdef]` 可以用 `[a-f]` 来代替。

字符集合中，第一个字符是`[^]`时，表示取反。就是说，不与括号 (`[]`) 中的字符相匹配。

字符集合中，想指定^、-、]时，如果开头不是^，则将-或]放在开头。另外，这些字符的前面如果放一个反斜杠（\）进行转义，就会作为普通字符来处理。

任意一个字符

表示任意一个字符的模式是“.”。但是，“.”不能匹配换行符（除了后面讲述的指定 m 选项的情况）。

比如说，以 P 开始，任意 2 个字符之后跟 l 的模式就是 P..l。“.”常常与下面要讲述的“重复”一起使用。

重复

模式的语法中也应有控制结构。正则表达式的控制结构是重复。根据上限与下限的不同，正则表达式的重复有多种写法（参见表 8-2）。每种都是对前面的模式进行重复。

表8-2 重复的写法

写 法	下 限	上 限
a*	0	无
a+	1	无
a?	0	1
a{n,m}	n	m
a{n}	n	n
a{n,}	n	无

比如说，a* 是指 a 重复 0 次以上，a+ 是指 a 重复 1 次以上。a? 是指 a 重复 0 次或 1 次，意思就是 a 或者无。

贪婪与懒惰

正则表达式中，有从最左边开始选择最长匹配的最左最长原则。但有些情况下，这很让人为难。

比如，为了提取字符串 <http://www.ruby-lang.org:80/ja> /开头的 `http:`，用 `.*:` 去匹配，结果却匹配到了 <http://www.ruby-lang.org> :。

原因就在于正则表达式中重复结构的贪婪（**greedy**），一定要找到与模式相一致的最长的字符串。在正则表达式的贪婪匹配过程中，即使遇到冒号也不会停止扫描，而是试图找到相匹配的最长字符串。

也就是说，继续检索冒号，一直到字符串的最后，发现“啊，没有了”再折回。所以，上面的就匹配到了最后一个冒号。这种折回称为回溯（**backtrack**）。

如果后面跟一个重复系列的元字符`?`，就不再贪婪，而是进行懒惰（**non-greedy**）匹配。懒惰匹配时，第一次与模式匹配时就停止检索。上面的字符串用模式 `.*:?` 匹配时，就能得到 `http:`。

分组

重复是以近前的正则表达式为对象的，`ma+` 是指 `m` 与跟随其后的 1 个以上的 `a`。如果想要表达 `ma` 重复 1 次以上，就要用括号括起来，变成 `(ma)+`。这种将模式绑定起来的功能称为分组。

与正则表达式中用括号括起来的模式相匹配的字符串，可以用 `\1` 等来表示。也就是说，`(.) \1` 是指连续两个相同字符。在后面使用匹配的字符串称为向后引用。

使用括号进行向后引用的时候，因为要在内部保存数据，程序效率多少有点降低。如果仅仅是为了分组，不需要进行向后引用的时候，就可以使用没有向后引用的模式括号 `(?:` 和 `)`。

选择

从多个模式中选出一种的元字符是 `|`，比如 `yes|no`。`|` 与其他正则表达式的构成要素相比，优先级要低，`yes|no` 可解释成“yes”或者“no”，而不会解释成“ye”后跟着“s”或者“n”，然后跟着“o”。那种模式可以使用分组来表达，写成 `ye(s|n)o`。

锚点

目前为止介绍的模式都是以字符的排列进行匹配的，正则表达式中，还有只用指定位置而不是字符来进行匹配的模式。这称为锚点（anchor）。锚点示意于表 8-3 中。

表8-3 锚点的例子

锚 点	匹配位置
^	行首
\$	行尾（若含换行，则是其前面字符）
\A	字符串头
\Z	字符串尾（若含换行，匹配前一字符）
\z	字符串尾
\b	词头或词尾（[]外）
(?=)	用模式指定位置
(?!)	用否定模式指定位置

作为练习，下面来解读以下正则表达式的意思吧。

正则表达式 1： `/FOO/`

这是指 FOO，也就是 F、O、O 三个字符排列起来的意思。

正则表达式 2： `/[A-Z][a-zA-Z1-9]*/`

以大写英文字母开始，紧接着是英文字母或数字，这是 Ruby 常数的模式。

正则表达式 3： `/(Ruby)+/`

Ruby 重复一次以上，能匹配 Ruby、RubyRuby、RubyRubyRuby..... 等字符串。

正则表达式 4： `/Dec,?/`

Dec 之后的“,”重复 0 次或 1 次，也就是与 Dec 或 Dec,相匹配。

正则表达式 5： `/Sun|Mon|Tue|Wed|Thu|Fri|Sat/`

与用英文表示的一周 7 天匹配。选择 `()` 的结合强度弱，匹配的对象不是近前的字符，而是字符串。

8.1.3 3 个陷阱

不仅是在 Ruby 中，在 Perl 及 AWK 中也是一样，经常有人说正则表达式很难懂。这是有理由的。除了已经讲过的它是语言中的微型语言之外，还有以下的理由。

记号多、密度高的表达式

正则表达式用记号来表达模式，看上去似有很多记号。像 Perl 一样，字符挤得密密的，看起来像“噪声”一样。而且，所有的字符都有含义，没有提供一种可以写成易读表达式的选项。为应对这一问题，出现了扩展正则表达式（后面会讲到）。

0 次以上的重复

写正则表达式时最容易在这儿卡壳。有人学了正则表达式很长时间，也写了很多正则表达式，到这儿还是容易出错。

比如说，表达“a 重复 0 次以上，后面跟着 bb”这个模式的 `a*bb` 与 `ccbbaabb` 进行匹配，会匹配上哪一部分呢？是 `aabb` 吗？但答案是前面的 `bb`。

0 个 a，后面跟着 bb，就变成了简单的 `bb`。正则表达式匹配字符串时，从左侧开始扫描，第一次匹配上时就停止了，后边即使有更适当的，也不会再往后检索了。如果要保证有 1 个以上的 a，可以写成 `a+bb`，但也不是每次都能保证有 1 个以上的 a。

贪婪型匹配

另一个容易卡住的地方是前面讲述的，正则表达式的扫描是贪婪的。我们已经说过，这可以用懒惰型匹配来解决。还有一种指定方法，将模式写成 `[^:]*:` 的样子，意指“冒号以外的字符重复 0 次以上，之后跟着冒号”。

8.1.4 正则表达式对象

面向对象语言 Ruby 中，所有数据都是对象。也就是说，正则表达式也是对象。Ruby 程序中正则表达式对象写成 `/.*` 的样子。

如果想写文件路径那种含有很多斜杠 (/) 的模式时，每次都写成 `/\usr(\local)?\bin/`，这样用了很多表示转义的反斜杠 (\)，就会搞得很复杂。

在灵活性方面很拿手的 Ruby，为这种情况准备了像 `%r!/usr(/local)?/bin!` 中的 `%r!` 这种正则表达式。`!` 可以是任意字符，应该成对出现。如果用括号，上式就变成了 `%r{/usr(/local)?/bin}`。

正则表达式对象可以用正则表达式类的类方法生成。程序中由组合字符串生成正则表达式时，使用类方法更自然。比如，**Regexp.compile** 接受字符串作为参数，生成相对应的正则表达式对象。

```
Regexp.compile("RE")
```

类方法参数带过来的字符串中，如果出现与元字符相同的字符，若不想让其表以特别的意义，可以对元字符进行转义，使其失去特别的意义。

比如，调用 **Regexp.escape**，为了消除其参数带过来的字符串中字符含有的元字符意义，该函数就会返回在字符串中加入了反斜杠的字符串。基于这个字符串做正则表达式，就能做出匹配 `P.*` 本身的正则表达式。

```
Regexp.escape("P.*")  
#=> "P\\.\\\\"
```

8.1.5 选项

Ruby 正则表达式末尾斜杠的后面，可以为这个正则表达式添加选项，如图 8-1 所示。

```
/ruby/i      # 不区分大小写
/a=#{a}/o    # 只展开一次
/あい/e      # 文字代码为EUC
/i あい/ei   # 可指定多个选项
```

图 8-1 选项的例子

可以指定的选项示于表 8-4 中。

表8-4 正则表达式的选项

选 项	含 义
i	不区分大小写
m	复数行匹配，换行看做普通字符
o	只展开一次
x	无视正则表达式中的空格，注释有效
e	假定是EUC字符串进行匹配
s	假定是SJIS字符串进行匹配
u	假定是UTF-8字符串进行匹配
n	看做是字节串进行匹配

使用 i 选项，则正则表达式匹配的时候，不区分字母的大小写。i 指 ignore case。

o 选项表示只展开一次。Ruby 的正则表达式和字符串中，以#{ }包起来的部分，可以填入任意表达式的值，这称为展开。正则表达式附带 o 选项时，展开只限于第一次。o 指 once。

m 选项表示多行匹配模式。通常，.不匹配换行符，^和\$也匹配字符串中途的换行符，但使用 m 选项，则将多行看做是一个整体字符串。

这样一来，不需要特别换行处理，就能跨越多行进行匹配。.也能匹配换行符，^也只能做到匹配字符串头，\$ 则只能做到匹配字符串尾。m

指 multi-line。

附加 x 选项后，可以在正则表达式中有意义的分隔处加上空格或注释。这种扩展正则表达式，比起难读的正则表达式来，可以写得明白易懂。比如 2007-05-03 这样的日期，可以用下面的正则表达式来匹配。

```
/\d{4}-?\d{1,2}-?\d{1,2}/
```

用扩展正则表达式，加上缩进和注释，可以写得更加易懂。

```
/\d{4}-?      # 年
 \d{1,2}-?    # 月
 \d{1,2}      # 日
/x
```

附加了 x 选项的正则表达式中想要匹配空格时，用\s。x 表示 extended。

剩下的选项 e、s、u、n 是指文字代码，分别是 EUC-JP、Shift_JIS、UTF-8、NONE（字节串）的意思。但 Ruby 1.9 里，每一个文件指定一种文字代码，几乎没有以正则表达式为单位使用这些选项的情况。

正则表达式对象的方法示于表 8-5 中，只有 6 个，其中还有 3 个是同一方法。结果，正则表达式对象的本质只有“匹配”这一点了。正则表达式的类方法示于表 8-6 中。

表8-5 正则表达式对象的方法

方 法	含 义
re === str	即match
re =~ str	即match
casefold?	是否不区分大小写
kcode	对应的文字编码
match(str)	匹配

source	正则表达式的字符串表示
--------	-------------

表8-6 正则表达式类的方法

方 法	含 义
compile(str)	正则表达式
escape(str)	元字符的转义
last_match	最后的匹配信息
new(str)	即compile
quote(str)	即escape

8.1.6 正则表达式匹配的方法

正则表达式匹配使用`=~` 方法或 `match` 方法。`=~` 方法在匹配成功时，返回代表匹配处位置的整数（字符串开头是 0），相当于以下代码中的第 1 行。

`match` 方法在匹配成功时，返回代表匹配信息的 `MatchData` 对象，相当于以下代码中的第 2 行。匹配不成功时，两种方法都返回 `nil`。

```
/P.*1/ =~ "Perl" # => 0
/P.*1/.match("Perl")
# => <MatchData:0x401b8>
```

匹配所使用的`=~` 方法在字符串类中也有定义，刚才所示代码像下面这样左右对调，也具有同样意义。

```
"Perl" =~ /P.*1/ # => 0
```

但有一点应当注意。当`=~` 两边都是字符串时，右边作为模式来解释。

```
"Perl" =~ "P.*1" # => 0
```

从 `match` 方法的返回值 `MatchData` 中，可以获取匹配位置、部分匹配（括号括起来的部分）、对象本身，以及匹配前后的字符串。部分匹配的下标指定为 0 时，可以取出匹配全体。

`MatchData` 类的方法示于表 8-7。实际使用例示于图 8-2。

表 8-7 `MatchData` 类的方法

方 法	含 义
<code>m[n]</code>	第n个括号匹配字符串
<code>begin (n)</code>	第n次部分匹配的开头
<code>end (n)</code>	第n次部分匹配的末尾
<code>length</code>	长度 (size)
<code>offset (n)</code>	第n次部分匹配的位置 (开头和末尾)
<code>post_match</code>	匹配部分之后的字符串
<code>pre_match</code>	匹配部分之前的字符串
<code>size</code>	部分匹配的字符串长度
<code>string</code>	匹配对象字符串

```
m = /a(.)(.)/.match("abc")
m[0]      # => "abc" (匹配全体)
m[1]      # => "b" (第1 个括号)
m.begin(0) # => 0 (匹配全体)
m.begin(1) # => 1 (第1 个括号)
m.offset(2) # => [2,2] (第2 个括号)
```

图 8-2 正则表达式的使用方法

8.1.7 特殊变量

Ruby 中有起源于 Perl 的特殊变量。以 \$ 开头的这些变量，有使程序变得丑陋的倾向，所以容易让程序员们的讨厌。但有一种好处，对于那

种只用一次，写完就扔的短程序而言，使用特殊变量可以使程序变得简短。与正则表达式相关的特殊变量示于表 8-8。

表8-8 与正则表达式相关的特殊变量

特殊变量	含 义
\$~	最后的MatchData （与regexp.last_match 相同）
\$&	最后的匹配字符串
\$`	位于匹配前的字符串
\$'	位于匹配后的字符串
\$+	匹配最后括号的字符串
\$n	第n个括号相对应的字符串（n为1， 2， 3等）

8.1.8 字符串与正则表达式

正则表达式的目的就在于匹配字符串，与字符串关联就是它的全部意义。Ruby 中字符串对象也使用很多的正则表达式。这里讲解字符串对象里使用的正则表达式。

与正则表达式相关的字符串对象的方法示于表 8-9。

表8-9 与正则表达式相关的字符串对象的方法

方 法	含 义
s =~ re	匹配
s[re]	获取字符串的一部分
s[re]=v	更新字符串的一部分
index (re)	部分字符串检索
gsub (re,str)	字符串置换
gsub! (re,str)	字符串置换
rindex (re)	从后面开始的index
scan (re)	循环匹配
split (re)	字符串分割
sub (re,str)	字符串置换
sub! (re,str)	字符串置换

[index] 与 [rindex] 是检索部分字符串的方法。可以指定正则表达式来代替部分字符串。“[]”与“[]=”是取出部分字符串的方法，可以用正则表达式来指定位置。

8.1.9 split 的本质

分割字符串的方法 `split`，与正则表达式组合起来能实现很多功能。

首先，用固定字符串（1 个字符）分割字符串时用下面的写法。

```
"a,b,c".split(",")  
# => ["a", "b", "c"]
```

同样，也可以用正则表达式来分割，以下是用,或者:来进行分割。当然可以用更复杂的模式。

```
"a,b:c".split(/[,:]/)    # => ["a", "b", "c"]
```

可以像下面这样用 **HTML**（或 **XML**）的标签部分来分割，以数组的方式取得标签包起来的部分。

```
str.split(/<.*?>/)
```

如果不想去掉标签部分，而是将其原封保存下来，可以像下面这样，用括号将模式包起来。因为用 `split` 方法，括号包起来的部分包含在数组中。

```
str = "<ul><li>a<li>b</ul>"  
str.split(/(<.*?>)/)  
# 分割成这种样子  
# ["", "<ul>", "", "<li>", "a", "<li>", "b", "</ul>"]
```

8.1.10 字符串的扫描

上面讲了 `split` 方法用正则表达式来分割字符串，`scan` 方法可以像下面这样，取出与字符串相匹配的部分。

```
"foo".scan (/./)
# => ["f", "o", "o"]
```

`scan` 方法的正则表达式中包含括号的时候，每个匹配就会追加一个数组，该数组由括号中匹配的字符串构成。也就是说，这时的 `scan` 方法返回由字符串数组构成的数组。

```
"fo".scan(/(.)(.) /)
# => [["f", "o"]]
```

`scan` 方法后面如果指定了用花括号（`{}`）括起来的程序块，则对于每一个匹配，都执行该程序块。

```
"foobarbazfoobarbaz".scan (/ba.){|s| p s}
# 输出以下行
# "bar"
# "baz"
# "bar"
# "baz"
```

下面代码执行同样的动作。它不生成无用的数组，效率稍高一点。

```
"foobarbazfoobarbaz".scan (/ba.){|s| p s}
```

8.1.11 置换

想要置换与字符串模式匹配的部分，可以用置换方法。置换方法有 4 种，根据用途分别使用（参见表 8-10）。

表8-10 置换方法

方法	复数置换	字符串的更新	不更新返回nil
sub	×	×	×
sub!	×	○	○
gsub	○	×	×
gsub!	○	○	○

置换方法中的 **sub** 方法，仅仅置换与模式匹配的最初部分。**gsub**（g指 **global**）方法置换与模式匹配的所有部分。方法名后附加!，表明置换字符串本身如果模式不匹配，则返回 **nil**；方法名后没有!，返回进行置换的字符串，原字符串不发生变化（参见图 8-3）。

```
a = "abcabc"
a.sub (/b/, 'B')
# => "aBcabc"(a 没变化)
a.sub! (/b/, 'B')
# => "aBcabc"(a 也更新)
a.sub! (/d/, 'D')
# => nil (因为不匹配)

a = "abcabc"
a.gsub (/b/, 'B')
# => "aBcBaBc"
```

图 8-3 置换的例子

几乎在所有的情况下，模式都不是去匹配固定字符串，而是要匹配各种各样的字符串。很自然，想根据匹配结果决定用什么字符串进行置换。对于这一要求，有下述两种对应方法。

一是使用表示置换字符串中匹配结果的元字符。在置换方法第 2 个参数指定的置换字符串中，**\&**、**\0** 是整个匹配部分的字符串，**\1** . . . **\9** 是第 **n** 个括号内匹配的字符串。

置换字符串中可以使用 **\`**、**\'** 和 **\+**。它们与 **\$`**、**\$'**和**\$+**相对应。

```
a = "abcabc"
a.sub(/[bc]/, '(\&)')
```



```
# => "a (b) cabc"
```

二是使用 `block`。省略置换方法的第 2 个参数（置换字符串），代之以 `block`。用 `block` 的执行结果来置换与正则表达式相匹配的部分。

```
"foobarbazfoobarbaz".gsub(/ba./) {|s|s.upcase}  
# =>"fooBARBAZfooBARBAZ"
```

置换字符串不能使用 `$1` 等特殊变量。因为对字符串进行运算那一刻，还没开始匹配。另外，用双引号（`"`）引起来的字符串中，如果有反斜杠（`\`），必须进行转义处理，所以建议使用单引号（`'`）。

用参数指定置换字符串时，常犯的错误示于图 8-4。

```
s = 'abbbcd'  
s.gsub (/a (b+) /, "#{ $1 }")  
# 错误  
s.gsub (/a (b+) /, "\1")  
# 错误  
s.gsub (/a (b+) /, "\\1")  
# 正确  
s.gsub (/a (b+) /, '\1')  
# 正确  
s.gsub (/a (b+) /, '\\1')  
# 正确  
s.gsub (/a (b+) /) { $1 }  
# 正确
```

图 8-4 常犯的错误

第一个错误在于式子展开是在调用 `gsub` 之前进行的，展开的是匹配之前的 `$1` 的值。

第二个错误在于用双引号（`"`）将置换字符串引起来。`\1` 被解释成 `\001`，也就是字节值为 1。

8.2 正则表达式的应用实例与“鬼车”

首先简单整理一下正则表达式。所谓正则表达式，是表达字符串模式的一种微型语言。正则表达式由字符本身、字符模式、锚点以及重复等组合而成。

比如说，`a` 是与字母 `a` 相匹配的正则表达式。`.` 是除换行以外能与任何一个字符相匹配的正则表达式。`*` 是指其近前的正则表达式重复 0 次以上。

像`.`、`*`那样，能够表达某种含义的字符称为元字符。正则表达式的元字符示于表 8-11。将元字符作为普通字符使用的时候，在其前面加反斜杠 (`\`)。

表8-11 正则表达式的元字符

元 字 符	含 义
.	与任意字符相匹配
[]	[a-z] 指a 到z 之间任何一个
[^]	[^a-z] 指a 到z 以外的字符
\w	构成单词的字符
\W	构成单词的字符以外
\s	空白文字。与[\t\n\r\f] 相同
\S	非空白文字
\d	数字。与[0-9] 相同
\D	非数字。与[^0-9] 相同
*	重复0次以上
+	重复1次以上
?	重复0次或1次
{m,n}	重复m次到n次
*?	重复0次以上（懒惰匹配）
+?	重复1次以上（懒惰匹配）
??	重复0次或1次（懒惰匹配）
{m,n}?	重复m次到n次（懒惰匹配）
^	行首匹配
\$	行尾匹配
\A	字符串头

\z	字符串尾（若含换行，匹配前一字符
\Z	字符串尾
\b	backspace(0x08)([]内)
\b	词头或词尾（[]外）
\B	非词头亦非词尾
	选择
()	分组（有向后引用）
\1,\2	向后引用（对应第n个括弧）
(?:)	分组（无向后引用）
(?=)	用模式来指定位置（无宽度）
(?!)	用否定模式来指定位置（无宽度）
(?#)	注释

将这些组合起来，就可以写出正则表达式的各种模式来。比如，下面这样，

```
a.*
```

就表示 a 的后面跟任意的文字。也就是说，a、ab 和 aabc 都能匹配。请注意，“*”是指重复 0 次以上，单单 a 也能匹配。

8.2.1 解析日志文件的方法

现在来看看正则表达式的使用。图 8-5 所示的是 HTTP 服务器 Apache 记录在访问日志里的信息。因为一行太长了，所以换行表示。

IP 地址	日期	HTTP 请求	返回码	文件长度
202.xxx.xxx.xxx	[07/Apr/2008:16:36:43 +0900]	"GET /~matz/20080323.html HTTP/1.0"	200	5563
7	"http://www.rubyist.net/~matz/20080323.html"	"Mozilla/5.0 [ja] (X11;I; Linux 2.6.18-5 i686		
	; rv:1.8.1.13) Gecko/20080311 Firefox/2.0.0.13"			
		访问源的 URL	浏览器信息	

图 8-5 Apache 访问日志的例子

在这个访问日志里，含有 IP 地址（为了保护个人信息，图中具体的 IP 地址以 xxx 置换）、日期、HTTP 请求、返回码、文件长度以及访问源的 URL 等信息。

这些信息，我们一看就能简单地理解，但若要以计算机容易处理的形式取出则很麻烦。这时候就该正则表达式显威力了。

首先取出 IP 地址。想一想 IP 地址应该具有怎样的模式吧。严谨一点说，就是由点（.）连起来的 4 个 1 到 255 之间的十进制数，也可以认为是由点连起来的 4 个数。这样，模式就成为下面这个样子。

```
\d+\.\d+\.\d+\.\d+
```

解释一下这个模式。`\d` 与数字匹配，`+` 是重复 1 次以上，`\.` 匹配点本身。如果考虑到数字的位数，可以指定重复的次数以取代`+`，成为如下的样子。

```
\d{1,3}\.\d{1,3}\.\d{1,3}\.\d{1,3}
```

具体程序如图 8-6 所示。图 8-6 的程序放在文件 `log1.rb` 中。

```
1  #! /usr/bin/ruby
2  ip = Hash.new(0)
3  ARGF.each do |line|
4    if /^(\d+\.\d+\.\d+\.\d+)/ =~ line
5      ip[$&] += 1
6    end
7  end
8  printf "%15s %s\n", "IP addr", "num"
9  ip.each do |ip,n|
10   printf "%15s %d\n", ip, n
11 end
```

生成哈希表，当访问不存在的元素时，返回 0

对命令行指定的文件的每一行进行匹配循环

图 8-6 对 IP 地址进行计数的日志解析程序

```
ruby log1.rb logfile
```

输入以上命令启动程序后，输出访问源的 IP 地址和来自该地址的访问次数。

说明一下图 8-6 所示程序的内容。第 2 行，生成了一个访问不存在的元素时返回 0 的哈希表。

第 3 行到第 7 行，对于由命令行参数指定的文件的各行进行循环操作。

循环中，对读入到名为 **line** 的局部变量中的各行，执行正则表达式的匹配。

Ruby 中，将正则表达式嵌入到一对斜杠 (/) 中来记述。**=~** 是匹配运算符。为了只匹配行首的 IP 地址，用了表示行首的正则表达式 **^**。

匹配成功的情况下，把哈希表中该 IP 地址对应的哈希值增加 1，进行计数。匹配的字符串放在变量 **\$&** 中，以此为键值操作哈希表。

这里，把哈希表的缺省值设置为 0 就起到了作用。用 += 运算符给指定的元素计数时，一开始是尚未定义返回 0，然后将其加 1，正好是该元素的计数值。

现在，提取日期信息，取出每个日期中访问最多的 3 个 URL 吧。

从图 8-5 的日志信息中看，日期是 [07/Apr/2008:16:36:43 +0900] 这样的形式。URL 的格式是 GET /~matz/20080323.html HTTP/1.0。

首先，来考虑一下匹配这个日志中所含日期形式的正则表达式。

日期是由两位数字、/、月的名字（3 个字符）、/，以及 4 位数字所构成。正则表达式就写成

```
\d{2}/[A-Z][a-z][a-z]/\d{4}
```

[A-Z] 是与 A 到 Z 之间的任意一个字符匹配的正则表达式，称为字符类。同样，[a-z] 是与小写字母匹配的正则表达式。

另外，访问对象可以用下式取出。

```
GET [^ ]+ HTTP
```

字符类中开头的`^`表示取反，所以，`[^]`是与空格之外的字符相匹配的字符类。

可是，这个正则表达式连前后的 **GET** 和 **HTTP** 也匹配了。但要不包含它们，可能把日志中别的部分也匹配了。这种情况下，将想要取出的部分用括号括起来，就可以只取出匹配的字符串。就像下面这样，

```
GET ([^ ]+) HTTP
```

匹配成功以后，括号内对应的字符串可以用`$1`等取出。只要记住这一点就行了，第 `n` 个括号内对应的字符串保存到`$n`中去了。

使用这些知识，编写取出每个日期中访问最多的 3 个 URL 的程序，如图 8-7 所示。

```
1 #! /usr/bin/ruby
2 date = {}
3 ARGF.each do |line|
4   if /\d\d\d\[A-Z][a-z][a-z]\d\d/ =~ line
5     date[$&] ||= Hash.new(0)
6     dic = date[$&]
7     if /GET ([^ ]+) HTTP/ =~ line
8       dic[$1] += 1
9     end
10  end
11 end
12 printf "%15s %s\n", "IP addr", "num"
13 date.keys.sort.each do |d|
14   puts d
15   date[d].sort_by{|k,v| -v}[0..2].each do |url,n|
16     printf "%-25s %d\n", url, n
17   end
18 end
```

} → 改变哈希表的结构

图 8-7 取出每个日期中访问数最多的 URL 的日志解析程序

其基本结构与图 8-6 的程序是一样的。一个区别是，为了按日期对 URL 的访问次数进行排序，更改了哈希表的结构：从“IP 地址 → 计

数”这种单纯的哈希表，变成“日期→{访问 URL→计数}”这种二重哈希与日期对应的哈希表，而这个哈希表中访问 URL 对应计数。另一个区别是，为了计算每个日期的访问数的前 3 名，对日期及计数进行了排序。

特别是，为了以计数为基础对哈希表进行排序，使用了 `sort_by` 方法（为了逆序，对计数的符号进行了反转），这个技巧在很多应用中都很有效。

使用正则表达式时应注意以下两点。

- `/` 是正则表达式的分割符（开始及结尾标志），虽然不是元字符，但用作普通字符时有必要以反斜杆进行转义。
- 取出匹配的字符串全体时用 `$&`；取出括号里对应的字符串时用 `$1`。

通过图 8-6 和图 8-7 所示程序的应用，从日志文件中提取出符合特定模式的行就变得轻松了。

8.2.2 避免使用\$的方法

但是，满是记号 `$&` 和 `$1` 的程序看起来可不怎么美观。Ruby 中，以 `match` 方法代替 `=~` 运算符，就可以在程序中不使用这些记号了。图 8-8 所示的程序是用 `match` 方法重写了图 8-7 的程序。

`=~` 运算符在模式匹配成功的时候，返回匹配位置（整数），而 `match` 方法在匹配成功的时候，返回 `MatchData` 对象，该对象中存放着所有关于模式匹配的信息。

另外，不管是 `=~` 运算符还是 `match` 方法，失败时都返回 `nil`。`nil` 看做是假，常作为一个惯用句放在 `if` 的判断条件里使用。

```

1 #! /usr/bin/ruby
2 date = {}
3 ARGF.each do |line|
4   if m = /\d\d\/[A-Z][a-z][a-z]\/\d\d/.match(line)
5     date[m[0]] ||= Hash.new(0)
6     dic = date[m[0]]
7     if m = /GET ([^ ]+) HTTP/.match(line)
8       dic[m[1]] += 1
9     end
10  end
11 end
12 printf "%15s %s\n", "IP addr", "num"
13 date.keys.sort.each do |d|
14   puts d
15   date[d].sort_by{|k,v| -v}[0..2].each do |url,n|
16     printf "%-25s %d\n", url, n
17   end
18 end

```

使用 match 方法

图 8-8 使用 match 方法的日志解析程序

如果将 **MatchData** 对象放入变量 **m** 中，则通过 **m[0]** 可获取匹配全体，通过 **m[n]** 可获取第 **n** 个括号所匹配的部分字符串。用 **m[1]** 取代 **\$1** 到底有多好，**\$** 有那么讨厌吗？等等疑问你或许也想过，但暂时先别管这些。

Ruby 1.9 更进了一步，在下面的条件全部满足时，部分匹配的结果会自动赋值给局部变量。这些条件是：正则表达式由常数字符串指定（不能有变量赋值）；用 **=~** 运算符进行匹配；使用带名字的部分匹配；带名字的部分匹配所指定的名字是有效的局部变量；该名字不是既存的变量。

使用 Ruby 1.9 的自动赋值的例子示于图 8-9。

```

if /(?!<first_name>\w+) \s+(?!<last_name>\w+)/ =~ "Yukihiro
Matsumoto"
  p first_name #=> "Yukihiro"
  p last_name  #=> "Matsumoto"
end

```

图 8-9 使用 Ruby 1.9 的自动赋值的例子

MatchData 类的方法示于表 8-12。

表8-12 **MatchData** 类的方法

方 法 名	含 义
m[n]	第n个括号匹配字符串
begin (n)	第n个部分匹配的开头
end (n)	第n个部分匹配的末尾
length	长度
offset (n)	第n个部分匹配的位置（开头和末尾）
post_match	匹配部分之后的字符串
pre_match	匹配部分之前的字符串
size	部分匹配的个数
string	匹配对象字符串

8.2.3 从邮件中取出日期的方法

前几天，看了 Mac OS X 的宣传录像，其中讲解了邮件的最新功能。点击邮件中看起来像日期的部分，日期就会自动排进日历计划中。对于计划多得像雪片一样的我来说，这真是令人羡慕的功能。

想想也是，我现在用自制的邮件阅读器，如果想要追加这样的功能也不是不可能。现在就挑战一下实现同样功能所需的核心部分——提取“看起来像日期的部分”吧。

看起来像日期，说得容易做着难。自然语言的日期表示并没有什么标准，表示形式多种多样。为便于说明，我们省去了时刻。从实用目的讲，时刻肯定也需要合适的表示形式。

首先，计算机行业使用的标准日期格式是“2009-06-01”，这是由 ISO-8601 标准规定的，对应于下面的正则表达式。

```
\d{4}-\d{2}-\d{2}
```

严格来说，公元前的年份应带负号。不过估计处理公元前也没有计划之事，先不管它了。

其次，可能会遇到日本式的日期表示形式“2009 年 6 月 1 日”，这种情况可以对应于下面的正则表达式。

```
[0-9 0-9]+年\s*[0-9 0-9]+月\s*[0-9 0-9]+日
```

因为要对应全角，所以模式有点长了。`\s*` 是为了匹配中途可能出现的空格。

要对应年号，年的前面加上下面这些应该更好了。

```
(明治|大正|昭和|平成|西曆|)\s*
```

当然，计算日期的时候需要加上各个年号的元年。

英文的日期表示形式太多了，让人无所适从。首先，年月日的顺序不一样。日本的日期表示形式几乎都是年月日的顺序，而美国的一般是月日年，欧洲的一般是日月年。

```
02.07.08
```

如果写成上面这个样子，到底是 2002 年 7 月 8 日（日本式），还是 2008 年 2 月 7 日（美国式），或者是 2008 年 7 月 2 日（欧洲式）呢？完全不能判断。哪怕有一个 4 位的公元年号也好，至少能判断出是哪一年。

Ruby 里有一个库 `parsedate`，在这方面做了不少工作。

```
require "parsedate"  
puts ParseDate.parsedate(str)
```

调用以上代码会返回“年、月、日、时、分、秒、时区和星期”的数组。但要注意，这也存在上述问题，表示并不完全。

另外，Ruby 1.9 已经不支持 `parsedate` 了，提供同样功能的是 `date` 库，请使用 `Date.parse` 方法。

8.2.4 典型拼写错误的检索方法

编辑文章时，重复助词（指日语中的“てにをは”）的情况时有发生。见过“私はは”这种错吗？其发生原因是这样子的。

比如“シンプルな文法”这句话，想把“シンプル”变成“簡潔”的时候，首先删除了“シンプル”，然后下意识地输入了“簡潔な”——连“な”也输入了，结果这句话变成了“簡潔なな文法”。这种意外的错误很多，我也给编辑带来了不少麻烦。图 8-10 为用于检查这类错误的脚本程序。

```
1 ARGF.each do |line|
2   print ARGF.file.path, " ",
3     ARGF.file.lineno, ":",
4     line if line.gsub!(/([へにおはがでな])\1/e, '[[\&]]')
5 end
```

图 8-10 检查拼写错误的脚本程序

这个脚本检查通过参数传过来的文件，如果遇到相同两个平假名并列在一起的部分，就用括号括起来并告诉你。“へにおはがでな”看起来虽然像咒语一样，根据经验，两个字符连接的地方最容易出错。以前曾用了“两个相同平假名连着”检查策略，但这样就把“ここ”、“くくる”的“くく”等（不该匹配的）也匹配上了。所以稍作了些限制。用 `([へにおはがでな])\1` 这种正则表达式，来表现括号内包含的平假名以及与其相同的字符并列。我的程序是用 EUC-JP 码写的，所以正则表达式的后面附带着选项 `e`，以明确表示用的是 EUC-JP 码。置换字符用的是表达式 `\&`，用以置换匹配的字符串。所以，用 `[[\&]]` 来指定并用 `[[]]` 把匹配全体都括起来。

还有就是，置换用字符串中的反斜杠（`\`），为了不让它有特殊意义，不要用双引号，推荐用单引号将其括起来。

图 8-9 所示脚本第 2 行、第 3 行中，用 `ARGF.file` 输出正在读入文件的路径和行号。`gsub!` 方法在模式匹配成功时，返回替换后的字符串，否则返回 `nil`，所以 `print` 只打印匹配成功的行。

8.2.5 Ruby 1.9 的新功能“鬼车”

到 1.8 版为止，Ruby 的正则表达式的库是基于为 Emacs 编辑器开发的库，使其对应 EUC-JP、Shift_JIS 和 UTF-8，并追加了与 Perl5 的互换功能。这个库虽足以应付日常的使用，但内部构造很复杂，无法再进行改善或功能扩展。

为了打破这一僵局，Ruby 1.9 采用了称为“鬼车”（oniguruma）的新正则表达式的库。“鬼车”的特征有以下 5 点。

- BSD License;
- 对应多种编码方式;
- Ruby1.8 的正则表达式库的上位互换;
- 高速;
- 增加了很多新功能。

到 Ruby 1.8 为止的正则表达式库与“鬼车”在语法上的差异示于图 8-11。

- 追加了文字Property 功能
- 追加了十六进制数字类型 (`\h`、`\H`)
- 追加了回读功能
- 追加了贪婪型循环用的元字符 (`?+`、`*+`、`++`)
- 追加了文字集合内的算符 (`[...]`、`&&`)
- 追加了带名字的捕获式集合以及部分式调用功能
- 文字集合中，可以指定 1 字节文字与多字节文字的范围
- 可以以普通字符串指定不完全循环的范围
- 追加了否定式 POSIX 方括号 (`[!^xxx:]`)
- 追加了 POSIX 方括号 (`[:ascii:]`)
- 不再许可先读循环
- 循环回数指定时，可以省略最低次数 (0 次)
- `/a{n}?` 不再是懒惰型算符

- 检查无效的向后引用，报告错误

图 8-11 Ruby 1.8 与 Ruby 1.9 正则表达式的差异

基于“鬼车”的新功能，之前无法实现的一些正则表达式匹配也能实现了。

以下的正则表达式可以用于匹配 **dad** 、 **eye** 这种从前读或从后读都一样的回文。

```
\A(?<a>|. |(?:(?<b>.)\g<a>\k<b+0>))\z
```

这样写密度太高了，不知道这是什么。用扩展正则表达式，加上缩进和注释重写就变成图 8-12 的样子。

```
\A          # 字符串先头
(?<a>       # 所谓回文
|.         # 或者空
|         # 或者1 个字符
|(?:(?<b>.) # 任意字符之后
\g<a>      # 发现回文（递归）
\k<b+0>))  # 与b 相同的字符
\z         # 字符串末尾
```

图 8-12 重写以后的回文匹配

在“鬼车”中，用 `(?<a>...)` 这种形式的表达式给正则表达式的一部分起名字，用 `\g<a>` 这样的表达式可以递归调用起了名字的正则表达式。这样，之前正则表达式不可能实现的嵌套功能（比如对应的括号等）也能实现了。

好好看看回文的正则表达式，会知道这正是回文的声明性定义。专家级程序员 **Dave Thomas** 对此很赞赏，感叹于“鬼车”所成就的函数式编程。

DSL

知道 DSL 这个用语吗？DSL 是 Domain Specific Language 的简称，意指面向特定领域的编程语言。这种技术根据某个特定领域的词汇和语法，不仅可以简化编程，提高生产性，而且有可能让非专业编程人员直接编写程序的逻辑。

DSL 有内部 DSL 和外部 DSL。所谓内部 DSL，就是往既存的语言里加入特定领域的词汇，使之 DSL 化。比如，软件编译工具 Rake 中表达依存关系的 Rakefile，就是记述软件编译中各种依存关系的内部 DSL。

```
task :default =>[:test]
task :test do
  ruby "test/unittest.rb"
end
```

上述 Rakefile 片段里，简洁表达了“缺省任务是 test”，“test 就是用 ruby 执行 test/unittest.rb”这样的关系。用 Ruby 实现内部 DSL，最优秀的一点是，即便增加了记述依存关系的词汇，但因为使用 Ruby，所以可以根据需要使用 Ruby 的全部功能。make 是一个同样目的的工具，它使用 Makefile 来表达依存关系，因为难于处理条件分歧和任务的模块化，关系一旦变复杂，记述也会变得非常困难。与 make 比起来，Rake 可以利用 Ruby 的编程功能、方法定义、条件分歧和循环等，不管关系有多复杂，都可以编程序对应。用脚本语言开发软件，为达成软件目的的词汇越加越多，因此内部 DSL 决定了软件开发的风格，这直接关系到软件开发的生产性。

所谓外部 DSL，不是扩展现有语言，而是面向特定目的，准备一种独自的语言。比如，访问数据库的 SQL（Structured Query Language，结构化查询语言）就是 DSL 一个有代表性的例子。外部 DSL 与 Ruby 这样的特定语言没有很强的结合关系，所以既能超越语言而共有知识，又能为着特定目的而进行优化。

本章所讲解的正则表达式也可以称作是以实现模式匹配为目的的外部 DSL。从一开始就嵌入到 Ruby 语言中，没给人留有 DSL 的印象。但正则表达式有单独一门语言那么复杂，这样想来，它有独立的语法，特定的目的，满足 DSL 的定义。

虽然 AWK 等脚本语言也与正则表达式这样长期共存，但从提供正则表达式的代表性语言 Perl 6.0 版开始，与正则表达式的关系就变

了。Perl 得益于其中称为 **rule** 的功能，语言本身具备了模式匹配，同时应用这些，提供了可扩展语言语法本身的功能。这可以看作是将外部 DSL 正则表达式吸收进来而进行的内部 DSL 化尝试。

另外，由内部 DSL 来表现 SQL 查询的方案也已登场，内部 DSL 与外部 DSL 的关系好像一直在变化。

不管是哪一种，DSL 都直接关系到生产性的技术，今后也有必要多加留意。

第 9 章 整数和浮点小数

9.1 深奥的整数世界

整数的基本操作有加、减、乘、除四则运算。编写程序所用的英文、数字和符号中没有×（乘）和÷（除），所以用* 和/ 来代替。以下是 Ruby 的例子。

```
puts 1+1 # => 2
puts 5-4 # => 1
puts 2*3 # => 6
puts 5/2 # => 2
```

请注意 $5 \div 2$ 的结果不是 2.5，而是 2。除法运算的结果，因编程语言的不同而有所不同。在 Ruby 中，整数运算结果只计整数，除法运算是舍去余数后所得的结果。计算余数的运算符是%。

```
puts 5%2 # => 1
```

对于数的操作，还有比较运算（参见表 9-1）。有了四则运算和比较运算，就可以写简单的计算程序了。试写一个计算阶乘¹的程序吧。

1 所谓阶乘（factorial）是指对于某个数值 n ，从 1 到 n 的所有整数的乘积。数学中 n 的阶乘用 $n!$ 来表示。 $4! = 24$ ， $1! = 1$ ，但 $0! = 1$ 。

表9-1 数的比较运算

比较运算符	含 义
==	等于
!=	不等于
>	大于
>=	大于等于
<	小于
<=	小于等于

用归纳法定义阶乘，就如下面这样。

若 $n=1$ ，阶乘为1
若 $n>1$ ，阶乘为 $n*((n-1)$ 的阶乘)

程序上大都采用归纳法的定义，使用递归函数调用² 来实现（参见图 9-1）。为了便于说明，程序用 C 语言来编写。程序的执行结果示于图 9-2 中。

2 所谓递归函数调用，是指在某函数 A 中再次调用 A 函数本身。

```
#include <stdio.h>
#include <stdlib.h>

int fact(int n){
    if(n == 1) return 1;
    return n * fact(n-1);
}

int main(int argc, char **argv){
    int i;

    for (i=1; i<15; i++){
        printf("fact(%d)=%d\n", i, fact(i));
    }
}
```



```
}
```

图 9-1 C 语言的阶乘运算程序，用了递归函数调用

```
fact(1)=1
fact(2)=2
fact(3)=6
fact(4)=24
fact(5)=120
fact(6)=720
fact(7)=5040
fact(8)=40320
fact(9)=362880
fact(10)=3628800
fact(11)=39916800
fact(12)=479001600
fact(13)=1932053504
fact(14)=1278945280
```

图 9-2 图 9-1 的程序的执行结果，计算结果有问题

仔细看看图 9-2，不觉得有什么不对劲吗？首先，**fact(13)** 的结果是 1932053504（1, 932, 053, 504），但不是 13 与 **fact(12) = 479001600** 相乘的结果。用 Ruby 来确认一下吧。

```
puts 1932053504/479001600 # => 4
```

而且，**fact(14)** 的结果要比 **fact(13)** 的结果要小。如果是数学中的整数，这样的结果显然让人想不通。

实际上，这正是计算机中整数的特征。

9.1.1 整数是有范围的

C 语言中有多种表示整数的数据类型（参见表 9-2）。每种数据类型能够表示的整数位数是一定的。位数是指二进制数据的位数。正如刚才所说的，二进制中的 1 位称作 1 比特，也可以说每种数据类型占据的比特数是不一样的。

表9-2 C语言的整数型

名 称	性 质	典型位数
char	能够表现ASCII码（英文、数字等）范围	8位
short	比int 小	16位
int	CPU最容易处理的范围	32位/64位
long	比int 大	32位/64位
long long	比long 大	64位

C 语言的标准中，只规定了 $\text{char} \leq \text{short} \leq \text{int} \leq \text{long} \leq \text{long long}$ 。极端一点，即使存在所有这些数据类型都具有相同位数的 C 语言处理系统，也不为过。16 位计算机上的 C 编译器中 `int` 一般是 16 位的，而有的超级计算机中 `short` 以上全是 64 位。通常，整数位数多如表 9-2 所示。

返回到图 9-2。C 语言中如果运算结果超出了规定的位数，不会报错，溢出的位仅仅被忽视了。阶乘计算程序出现异常的原因就在于此。

图 9-1 所示的程序中用 `int` 表示整数。我执行这个程序使用的 C 处理系统（Intel Core 2Duo，gcc 4.3.3），`int` 是 32 位。32 位能够表示的最大正整数是 $(2^{31}-1)^3$ ，也就是 2147483647(2,147,483,647)。

3 为什么 32 位能表示的最大正整数不是 2^{32} ，在 9.1.5 节中有说明。

12 阶乘的结果还在此范围内，但 13 的阶乘 6227020800（6,227,020,800）已经超出这个范围。所以从 `int` 溢出，结果就变得很奇怪了。

使用像 C 语言这类整数位数有限制的语言时，必须注意能够表示的数值范围。

实际上整数范围的问题，是连专家也可能会出错的难题。

比如，在 Linux 等 UNIX 系列操作系统中，时刻是以从 Epoch 到现在为止的秒数来表示的，Epoch 是一个特定的时间（世界协调时间 1970 年 1 月 1 日凌晨 0 时 0 分），但存放这个秒数的是 32 位整数，所以到

2038 年 1 月 19 日 12 时 14 分 7 秒（日本时间），32 位带符号整数所能表示的最大整数就到头了。如果像现在这样，2038 年以后的时刻就不能表示了。

在开发 UNIX 的 20 世纪 60 年代末到 70 年代初，2038 年还是很遥远的未来，而 30 多年过后如今已经离我们越来越近了。估计到 2038 年，时刻表示将扩展为 64 位整数，或是 128 位了。时间的表示是这种软件最基本的部分，升级起来相当困难，我担心 30 年后会像 2000 年问题那样引起喧嚣。

9.1.2 尝试位运算

加减乘除和比较是算术中的基本运算。与此对应，计算机中还有几个特有的整数运算，都是利用整数在计算机中以二进制来表示这一性质，称为位运算（参见表 9-3）。按位或、按位与以及按位异或，对构成整数的各位进行按位运算。

表9-3 位运算符

运算符	含义（简称）
	按位或（bit or）
&	按位与（bit and）
^	按位异或（bit xor）
~	按位取反（bit negate）
<<	左移（left shift）
>>	右移（right shift）

按位运算在日常生活中几乎不使用，来复习一下吧。

按位或，两边只要有一个是真，结果就是真。按位与，两边都是真时，结果才是真。按位异或，两边有一边且只有一边是真时，结果是真。各种按位运算的例子示于图 9-3。

按位或			按位与			按位异或		
	0	1		0	1		0	1
0	0	1	0	0	0	0	0	1
1	1	1	1	0	1	1	1	0

图 9-3 按位或、按位与以及按位异或的计算例子

图 9-3 显示了对两个值进行按位运算，会得到什么结果。上端和左端是输入，线框内是该组合所对应的演算结果。解读此图，按位或的结果如下所示。

```
0 | 0 = 0
0 | 1 = 1
1 | 0 = 1
1 | 1 = 1
```

进行位运算时，这种按位运算在二进制数的每一位中进行。举个例子，试计算 $201 | 5$ 。201 用二进制表示就是 11001001，5 用二进制表示为 101，运算靠右对齐。按照图 9-3，每一位边看边写结果。该位不存在则看做 0。

```
  11001001
|)    101
-----
  11001101
```

按位或的结果是二进制的 11001101。变成十进制就是 205。对人来说，计算按位或很麻烦，但计算机却最拿手。

```
puts 201 | 5 # => 205
```

别的运算也一样。

```
puts 201 & 5    # => 1
puts 201 ^ 5    # => 204
```

按位非则是将每一位求反：1 变成 0，0 变成 1。

还有按位运算以外的运算，那就是移位。

移位运算将数看做是一个二进制的 0 和 1 序列，向左或向右移动来进行计算。右移运算符是>>，左移运算符是<<。

来看看实际的移位运算处理吧。以 5 为例，5 以二进制表示是 101。向左移两位就是 10100。以十进制表示则是 20。反过来，20（10100）向右移一位结果变成 1010，十进制则是 10（参见图 9-4）。

```
puts 5<<2 # => 20 (= 5*4)
puts 20>>1 # => 10 (= 20/2)
```

图 9-4 移位运算的例子

根据二进制的性质，左移一位相当于将数变成 2 倍，右移一位相当于将数变成一半。这与十进制数移动一位变动 10 倍是一样的。

9.1.3 操作特定的位

位运算组合起来，可以对存储在计算机中的各位进行自由操作。

计算机中处理的各种数据都是以二进制的位来表示的。文本是由字符所对应编码的数字串组成，各数字串最后还是还原成二进制的比特。图像数据是各个点的颜色按一定形式排列，最后也还是二进制的比特串。程序所处理的对象，甚至计算机所执行的程序本身，说到底都是二进制的比特串。

也就是说，操作二进制位就等于操作计算机的数据。

基本的位处理操作有 4 种。

1. 取出特定位的状态。
2. 特定位置位（设为 1）。
3. 特定位置零（设为 0）。

4. 特定定位反转。

要从一连串二进制位中取出某一特定定位 a 的状态，该怎么办呢？当然，眼睛一看就能知道，但想想用程序的方法取出来。

这个功能可以用按位与（&）来实现。按位与的两方之中，只要有一方是 0，结果就必然是 0；一方是 1，另一方的结果就原封不动返回。所以准备一个数 B ，只需将想取出的那一位置设为 1，其余位置设为 0，那么取 A 与 B 的按位与，就可以只取出 A 数中 a 位的状态（参见图 9-5）。



图 9-5 取出二进制数 A 中特定定位 a 的状态的方法

仅生成某一特定定位为 1 的数，可以用移位运算符。用式 $1 \ll n$ ，就可以得到仅某一特定定位为 1 的数⁴。

4 位操作时，第 n 位是从最后一位开始数。按 C 语言习惯，最低位为第 0 位。

像 B 那样，将操作限制在特定定位的数称为掩码。

为了取出整数 A 第 3 位的状态，可以用下面的写法。

```
A & (1<<3)
```

位操作不仅可以取出信息，还可以更新。操作特定定位的时候，如果改变了该位以外的信息，可就麻烦了。将操作限制在特定定位，还是要用掩码。

比如，将变量 A 的第 3 位设为 1，用按位或写成以下形式。

```
A |= (1<<3)
```

比如 A 是 101（二进制），该式就变成

```
A = 0b101 | 0b1000  
A = 0b1101
```

以下，为了明确区分二进制数与十进制数，在二进制数的前面加上 0b⁵。

5 0b 中 b 是英文二进制 binary 的首字母。C 语言中没有数的二进制表示，Ruby 中这样表示二进制。

用按位与，可以不管原数内容，而将其某一位设为 1。

位清零，也就是将某一特定位设为 0，比设为 1 要麻烦些。要用到掩码、按位与及按位取反的组合。要将第 2 位清零的话，就是下面这个样子。

```
A &= ~(1<<2)
```

首先，用按位取反做了一个将特定位清零所需要的掩码。计算与此掩码的按位与就能实现清零操作。假设 A 中放 0b101，A 是 8 位。就变成

```
A = 0b101 & 0b11111011  
A = 0b001
```

最后是位反转，也就是 1 变 0、0 变 1。这与置位（将某位设为 1）几乎相同。用按位异或取代了按位或。

```
A ^= (1<<3)
```

编程中频繁使用的位操作，最常见的例子是标志位操作。用位操作，可以将多个标志位（指定 **ON** 或者 **OFF** 的选项）用一个参数来传递。图 9-6 是以 C 语言记述的用位操作实现的标志位操作的例子。

```
/* regexp.h */
/* 匹配不区分大小写 */
#define REG_OPTION_IGNORECASE (1L)
/* 可以使用perl 风格的扩展模式 */
#define REG_OPTION_EXTENDED (REG_OPTION_IGNORECASE<<1)
/* 换行符可以用 . 匹配 */
#define REG_OPTION_MULTILINE (REG_OPTION_EXTENDED<<1)
/* ^和$忽视换行 */
#define REG_OPTION_SINGLELINE (REG_OPTION_MULTILINE<<1)
/* 检索最长匹配, 像POSIX regexp 一样 */
#define REG_OPTION_LONGEST (REG_OPTION_SINGLELINE<<1)
struct regexp *reg_compile_pattern(char* str, int option);

/*使用例*/
re = reg_compile_pattern(pat,
REG_OPTION_IGNORECASE|REG_OPTION_SINGLELINE);
```

图 9-6 C 语言中标志位操作的例子

图 9-6 中，调用假想的正则表达式匹配函数。末尾的 `reg_compile_pattern()` 函数返回一个指针，指向对应于参数传递过来的正则表达式的字符串，编译以后得到的正则表达式结构。第 2 个参数是指定编译选项的标志位，用头文件中定义的常数（掩码）来指定。想指定多个标志位时用按位或。一个标志位都不指定时，第 2 个参数为 0。

9.1.4 表示负数的办法

8 位的整数可以表示 0~255 之间的数，但这样就不能使用负数了。怎么用二进制来表示负数呢？

有多种方式可以用来表示二进制负数。

- 开头一位用作符号位。

- 将整数的各位反转（1 的补数）。

8 位整数的情况，-1 用前一种方法表示是 10000001，用第 2 种方法表示是 11111110。

但负数一般用 2 的补数表示。所谓 2 的补数，是将正数的每一位反转，然后加 1 所得的数。比如，-1 的 2 的补数，可以用以下步骤来计算。

```
00000001  (数1)
11111110  (位反转)
11111111  (加1)
```

以 2 的补数来表示其他负数，会觉得有些麻烦，但有几个重要的优点。

- 可以表示 256 个数。采用符号位方式或 1 的补数方式，0 和 -0 都存在，只能表示 255 个数。2 的补数方式没有这种浪费。
- 可以直接运算。2 的补数方式，不考虑符号，直接进行四则运算就可以得到正确结果。

特别是后一种特性，成为 2 的补数方式被采用的最大理由。采用 2 的补数方式，8 位时能表示 -128~127。同样，16 位时能表示 -32 768~32 767，32 位时能表示 -2 147 483 648~2 147 483 647。通常， n 位时，2 的补数方式能表示 $-(1 \ll (n-1)) \sim (1 \ll (n-1))-1$ 之间的整数。

9.1.5 Ruby 的整数

学了位操作，再回到阶乘的话题。图 9-1 的程序是用 C 语言写的，下面使用 Ruby 来写。图 9-7 是 Ruby 的阶乘程序。

比较图 9-1 的程序（C 版）与图 9-7 的程序（Ruby 版），除了没有 main 函数、括号以 end 代替等外观上的区别以外，并没有什么读起来不懂的区别。

```
def fact(n)
```

```
    if n == 1
      return 1
    else
      return n * fact(n-1)
    end
  end
end

for i in 1..15
  printf "fact(%d)=%d\n", i, fact(i)
end
```

图 9-7 Ruby 的阶乘计算程序

那么执行一下试试吧（参见图 9-8）。

```
fact(1)=1
fact(2)=2
fact(3)=6
fact(4)=24
fact(5)=120
fact(6)=720
fact(7)=5040
fact(8)=40320
fact(9)=362880
fact(10)=3628800
fact(11)=39916800
fact(12)=479001600
fact(13)=6227020800
fact(14)=87178291200
fact(15)=1307674368000
```

图 9-8 图 9-7 程序的执行结果

Ruby 版的执行结果示于图 9-8，与图 9-2 的 C 版执行结果比较一下发现，**fact(13)** 以后的结果不同。为什么 Ruby 版能得到正确的阶乘结果呢？

C 语言中，能表示的整数范围有限制，演算结果溢出了，既没警告也没报错。而 Ruby 中能表示的整数范围没有限制。

Ruby 的整数有两种，一种是范围有限的整数 **Fixnum**（32 位 CPU 是 31 位，64 位 CPU 是 63 位），另一种是范围没有限制（超过内存容量

除外)的整数 **Bignum**, 根据计算结果自动变换。因为有此功能, **C** 中不能正确计算的 13 以上的阶乘, 在 **Ruby** 中能正确计算。**Ruby** 可以计算非常大的数, 用 **fact()** 函数, 可以计算 100 的阶乘 (参见图 9-9)。**100** 的阶乘用十进制表示是 158 位。可以毫不费事地进行这种计算, 是 **Ruby** 的整数的特长。

```
9332621544394415268169923885626670
0490715968264381621468592963895217
5999932299156089414639761565182862
5369792082722375825118521091686400
000000000000000000000000000000
```

图 9-9 用 **Ruby** 计算 100 阶乘的结果

Bignum 在内部, 分别保存符号和绝对值, 绝对值以整数数组形式存放。数组的各个元素是 32 位无符号整数⁶, **Bignum** 的内部表示中可以看做是 4294967296 进数。

⁶ 实际数组元素的位数因 CPU 及处理系统而有所不同。

Bignum 中符号另外保存, 与 **Fixnum** 不同, 内部没用采用 2 的补数, 但位运算在外表上看起来像是采用了 2 的补数。对于 **Ruby** 的位运算, 负整数的左侧看起来是无限多的 1。所以, **Ruby** 中如果像下面这样写:

```
printf "%x\n", -4
```

就会得到

```
..fc
```

这样谜一般的字符串。这是因为左侧排列着无限的 1, 所以 (在十六进制时) 就表示为无限的 f。

9.1.6 挑战公开密钥方式

关于计算机中的整数，我们讲解了四则运算和位处理，最后稍微介绍一下整数的算法。

数学中，整数是从古代就存在的概念，所以有很多古老的算法。比如称为最古老算法的欧几里得算法。据说，欧几里得生于公元前 330 年，的确很古老吧。图 9-10 是他发明的用欧几里得算法计算两个数的最大公约数的函数。

```
def gcd(x, y)
  if y == 0
    return x
  else
    return gcd(y, x % y)
  end
end
```

图 9-10 用欧几里得算法计算最大公约数

另一个很古老的算法是判定素数的埃拉托斯特尼筛选法。素数是指只能被 1 和其本身整除的数。据说埃拉托斯特尼生于公元前 275 年，这当然也是很古老的算法。

图 9-11 是用埃拉托斯特尼筛选法计算 100 以下素数的程序。

```
sieve = []
max = 100
for i in 2 .. max
  sieve[i] = i
end

for i in 2 .. Math.sqrt(max)
  next unless sieve[i]
  (i*i).step(max, i) do |j|
    sieve[j] = nil
  end
end

puts sieve.compact.join(", ")
```

图 9-11 用埃拉托斯特尼筛选法计算 100 以下素数

这个算法很简单，用 **sieve**（筛子）这个数组记录被判定为素数的数和其倍数。没有筛出的数就是没有约数的数（素数）。

要说整数领域的研究已经很完美，没有新的算法出现，完全没那回事儿。比如公开密钥就是利用整数性质的新算法。公开密钥算法的代表 **RSA**，发明于 1977 年，跟欧几里得与埃拉托斯特尼比起来，完全是现代的话题。

所谓公开密钥加密具有如下的性质：用公钥加密的字符串只有用私钥才能解读；反之，用私钥加密的内容只有用公钥才能解读。公钥密码加密，既可实现认证（自己确实拥有密钥的证明），又可实现加密（制作密文）。

简要介绍一下公钥密码加密的原理。假设有两个素数 p 和 q 存在，从这两个数计算图 9-12 中列举的数。

```
pq    = p × q
k     = n × (p-1) × (q-1)+1, n 为任意正数
e, d = 能使 e × d = k 的任意正数
```

图 9-12 公开密钥加密的原理

比如， $p=3$ ， $q=11$ ，就成为图 9-13 所示的样子。这里 (pq, e) 是公钥， (pq, d) 是私钥。图 9-14 是将密码及消息（整数数组）进行加密的程序。实际的公钥密码加密程序也是将文本先变为整数数组，然后再进行加密。

```
pq    = 33
k     = 1 × (3-1) × (11-1)+1 = 21
e, d = 3, 7
```

图 9-13 公开密钥暗号的例子

```
def rsa(pq, k, msg)
  msg.collect{|x|
    x**k%pq
  }
end
```

图 9-14 公开密钥加密的程序

公钥密码加密听起来很难，事实上只要准备了密钥，如果不用考虑效率，程序就简单得有点让人失望。那么，实际加密以后，再解读看看吧（参见图 9-15）。以公钥 3 加密的密文，再用私钥 7 来解密，就能得到原来的文本。

```
orig = [7, 13, 17, 24]
encode = rsa(33, 3, orig)
# encode => [13, 19, 29, 30]
decode = rsa(33, 7, encode)
# decode => [7, 13, 17, 24]
```

图 9-15 加密与解密的步骤

此例中，已经知道 33 是两个素数的积，马上得出 $p=3$ ， $q=11$ 。知道这些，很快就能计算 3 对应的私钥是 7。

但当 p 与 q 是非常大的素数时，从积 pq 计算素数（素因数分解）并不简单。RSA 暗号中一般使用的钥长（ pq 的位数）是 1024 位（二进制）。1024 位长的整数，用几千台超级计算机满负荷运行进行分散处理，在现实时间内，也难以进行素因数分解。RSA 加密的强度（解读的困难程度），就归因于素因数分解的难度。

9.2 扑朔迷离的浮点小数世界

刚进小学的时候，算术中学到的数是整数，而且仅有正整数。升到了高年级，小数登场了，像 0.2、1.5 等。到了中学，数的范围更扩大了，小数被认为是实数的一种。

9.2.1 计算机对小数的处理

计算机也能处理小数。程序中用含有小数点的数表示小数。puts 0.2 这句 Ruby 程序表示，生成 0.2 这个小数，然后输出。Ruby 中的所有数据都是对象，所以小数也是对象。表示小数对象的类是 Float

。本来是数，起了 float（漂浮）这么个奇怪的名字。计算机中的小数被称为浮点数（floating point number），由此得名。

9.2.2 固定小数点数不易使用

所谓浮点数，是指小数点的位置可以移动。它与计算机中数的表示方法有着密切的关系。

计算机本来只处理整数，事实上整数也是二进制的比特串。所以，必须要用某种方法，将含有小数部分的数变成二进制的比特串（编码）。

可以考虑用几种方法将小数编码成二进制。具代表性的方法有两种：一是抬高小数进行整数化；二是使用科学计数法来表示。

抬高小数是指将小数放大，比如说 100 倍¹，进行整数化，来表示小数点以下部分的方法。放大 100 倍的方法，能表示小数点以下两位。这种小数表示方法称为固定小数点数（fixed point number）。这种方法有时会用到，用于处理明确知道小数点后的有效位的数。

1 固定小数点数的表示不用十进制，用二进制的情况也很多。这种情况下，放大 2 的 n 次方倍（比如 256 倍）。

这种用整数运算的方法计算小数有速度高的优点，当然也有缺点。假设抬高两位（十进制，即放大 100 倍），那么 1 就成为了 1.00，小数点后的两位就浪费了。结果在位数一定的情况下，本来能够表示的数，现在只能表示较小的数了（整数部分的浪费）；十进制两位的情况，只能表示以 0.01 为单位的小数（小数部分的限制）。

因为这些理由，固定小数点数的使用不怎么广泛。

9.2.3 科学计数法也有问题

计算机中广泛使用的小数表示方法是科学计数法。科学计数法是指将有效数字和指数组合起来表示小数（实数），写成 2.5×10^4 ，这就是 25000。

但计算机是用二进制，而不是用十进制来表示数，所以实际上这个数在内部不是以 2.5 和 10^4 来记录的，必须变为二进制的比特串。

变为比特串的方式有多种。以下介绍广泛采用的 IEEE754 方式。

IEEE754 方式中，为了表示小数，单精度（float）用 32 位，双精度（double）用 64 位。有很多 CPU 在计算浮点小数时，内部用双精度²进行计算（单精度时，将双精度计算结果变成单精度）。以下只说明双精度。

² 更精确的有双精度以上的计算。比如奔腾以后 X86 系列 CPU 用 80 位长浮点数用于内部计算。

表示双精度 64 位比特串示于图 9-16，使用 $\pm f \times 2^e$ 的形式。

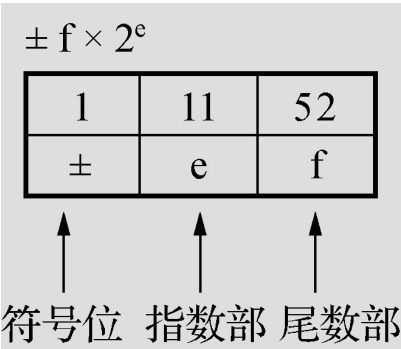


图9-16 IEEE754中double型浮点数的内部表示

f 称为尾数部分（mantissa），e 称为指数部分（exponent）。双精度中，指数部分有 11 位，可以表示 $+1023 \sim -1024$ ，也就是可以表示 2 的 1023 次方。

尾数部分有 52 位。IEEE754 规定，尾数部分的首位始终归一化³ 为 1，所以首位始终省略⁴，实质有效数字为 53 位。

³ 比如想表示 48，可以用 3×2^4 表示。所谓归一化就是将尾数部分变成大于等于 1，小于 2 的数，写成 1.5×2^5 的形式。

⁴ 因为归一化而被省略的位通称省略位。

9.2.4 小数不能完全表示

但是，使用以上的比特串，小数还是不能完全被表示，存在如下两个限制。

- 计算机中数的表示有长度（位数）限制。
- 计算机中数的表示是二进制。

这与之前讲述的计算机上处理整数时的限制没有差别。但就小数的情形，有更复杂的事情。

一是存在除不尽的数。比如在计算器上输入 $1 \div 3$ ，马上就知道，即使很简单的计算也会有除不尽的数出现⁵。

5 Ruby 中的有理数类（**Rational**）可以正确表示数学上的有理数（能表示为分数的数）。

```
p 1.0 / 3.0  
# => 0.3333333333333333
```

还有，存在圆周率 π （约 3.14）、自然对数的底 e （约 2.72）这种无限不循环的无理数。这种无限的数不可能放在有限领域内（舍入误差）来探讨。

更复杂的是，浮点数在计算机内部是以二进制表示的，就是能以十进制除得尽的数，在二进制也是除不尽的。比如说，小数 0.2 是 1 除以 5 所得的结果，十进制中能除尽，但二进制中是循环小数（0.0011001100...）。

```
p 0.2  
# => 0.2
```

Ruby 中能表示为 0.2，是因为 **double** 精度高。与实际值 0.2 足够接近的数，可以表示为 0.2。

总而言之，浮点数其实是真实数的近似，所以产生了限制。以下几点容易忘记，一定要时时注意。

浮点数是有限的

数学上有的数有无限多位，但浮点数只能拥有有限信息。单精度的浮点数只有 32 位，双精度的只有 64 位。用这么多位表示出的数也是有限的。

浮点数有误差

这与浮点数的有限性密切关系。任意的实数，只要是用有效数字和指数的组合来近似表示的，有效数字的不足部分被简单地忽略。多次运算后，与实际值的差（误差）就会积累起来，计算结果与理论值偏差很大是常有的事。误差的产生有很多形式，有时会让人陷入意想不到的陷阱。

对于浮点小数，结合法不成立

结合法是指加法和乘法中，不管计算顺序怎样，计算结果都相同的法则。对于数学上的数，这个法则在实数范围内都成立，但在浮点小数中就不成立。来看一个具体的例子。

$$(a + b) \times c$$

在这个式子中， $(a + b)$ 所产生的误差有被扩大的可能。要得到同样的结果，可以写成以下形式，误差会变小。

$$(a \times c) + (b \times c)$$

9.2.5 有不能比较的时候

虽说 Ruby 中某一浮点数可以用“比较整”的小数来表示，但其内部表示却不一定“比较整”。Ruby 里，表示出“1.0”，只是因为“聪明的”输出程序判定此值与 1.0 足够接近而已。

所以，同样表示为 1.0 的两个值，并不能断定是否真的是同一个值。来看一个例子。

图 9-17 所示为一个含有小数计算的简单程序。变量 **one** 的值为浮点数 1.0，变量 **sum** 的值为 10 个 0.1 相加所得的数。输出两个数的值都为 1.0。

```
one = 1.0
sum = 0.0
10.times do
  sum += 0.1
end
p one ==> 1.0
p sum ==> 1.0
p one == sum ==> false(!)
p one - sum ==> 1.11022302462516e-16
```

图9-17 浮点数运算的例子，0.1相加10次不能变成1.0

比较两个数，结果却是 **false**。看起来相同的两个数实际上并不一样。

两个数相减，所得的差是 1.11022302462516e-16。写成常见的形式就是 0.00000000000000011102230462516。0.1 加 10 次就产生了这么大的误差。

这与 0.1 是用二进制表示的循环小数有关。假设计算是以用二进制能除尽的 0.5 来加 10 次，就不会产生误差了。如果不知道浮点数的内部表示，就不可能理解这一行为。

对浮点数进行比较运算，只有两个数在内部表示是完全相同的情况下才判定为相等。作为铁则，两个浮点数不能用 **==** 进行比较。如果有进行比较的必要，判断条件中两个数的差要写得足够小。足够小是个很难的条件，根据处理系统的不同，对于浮点数，足够小的值 ϵ 有不同的定义。Ruby 中是 **Float::EPSILON**，其值为 2.22044604925031e-16。如果差比这个值还小，可以认为是舍入误差的积累。这个例子中，误差是 1.1102230462516e-16，比 ϵ 小，所以将这两个数判定为相等也行。

9.2.6 误差积累

图 9-17 所示的程序中，两个 1.0 不相等就是因为误差积累。同样是计算 0.1 的 10 倍，如果用乘法运算，这个问题就不容易发生了（参见图 9-18）。

```
one = 1.0
mul = 0.1 * 10
p one ==> 1.0
p mul ==> 1.0
p one == mul ==> true
p one - mul ==> 0.0
```

图 9-18 浮点数运算的例子，0.1 的 10 倍与 1.0 相当接近

图 9-17 中，二进制不能除尽的数 0.1 相加的结果也是除不尽的数，所以就在双精度的表示范围内进行舍入。计算后舍入，重复 10 次，误差就积累起来了。乘法运算只对结果舍入 1 次，与 10 次加法相比，误差要少得多。由此导出第 2 条铁则：减少运算次数。

9.2.7 不是数的特别“数”

IEEE754 中，除了普通的数，还定义了几个特别的数。不是数的数，那是什么呀？

这些“数”用于不同于普通数的目的。

无限大

为了表示在浮点数范围内无法表示的大数，提供了无限大（Inf）这个特别的数。无限大用于表示溢出错误。C 语言中，整数运算发生溢出后，运算结果会变成某一适当的值，而不产生错误。IEEE754 方式下，发生溢出时，不是将结果变成某一适当的值，而是使用无限大来表示溢出错误。

零

零是常见的数，但在 IEEE754 中进行了特别处理。IEEE754 中，零有符号，正零和负零要区别对待。可能是在除以零的时候，为了将结果区分为正无限大或是负无限大。

NaN

NaN 是 Not a Number（非数）的缩写。说是浮点数，却又是非数，很奇怪啊。NaN 作为结果赋给没有定义值的运算。比如零除以零，结果就是 NaN。一般认为，无限大是为了表示溢出错误，而 NaN 是为了表示未定义的结果错误。

NaN 不是正常值，含 NaN 的运算结果依然是 NaN。包括 NaN 自身，NaN 与任何数都不一致。

包含这些特别值的运算结果总结在表 9-4 中。

表9-4 含有特别数的运算结果

运算	结果
$n \div \pm \text{Inf}$	± 0
$\pm \text{Inf} \times \pm \text{Inf}$	$\pm \text{Inf}$
± 0 以外 / 0	$\pm \text{Inf}$
$\text{Inf} + \text{Inf}$	Inf
$n + \text{Inf}$	Inf
$n - \text{Inf}$	$-\text{Inf}$
$\pm 0 \div \pm 0$	NaN
$\text{Inf} - \text{Inf}$	NaN
$\pm \text{Inf} \div \pm \text{Inf}$	NaN
$\pm \text{Inf} \times 0$	NaN
含NaN的运算	NaN

9.2.8 计算误差有多种

浮点数的运算会产生误差。但同为误差，还分为几种。

舍入误差

循环小数及无理数等有无限多小数位的数，用位数有限的浮点数不可能完全表示，必须从某一位舍去。而且，因为内部表示是二进制，十进制中看起来能除尽的数，往往在二进制中是循环小数。想一想，十进制的小数 $0.n_1n_2\dots$ 表示的是图 9-19a 所示的内容。当然，二进制的小数 $0.n_1n_2\dots$ 表示的是图 9-19b 所示的内容。

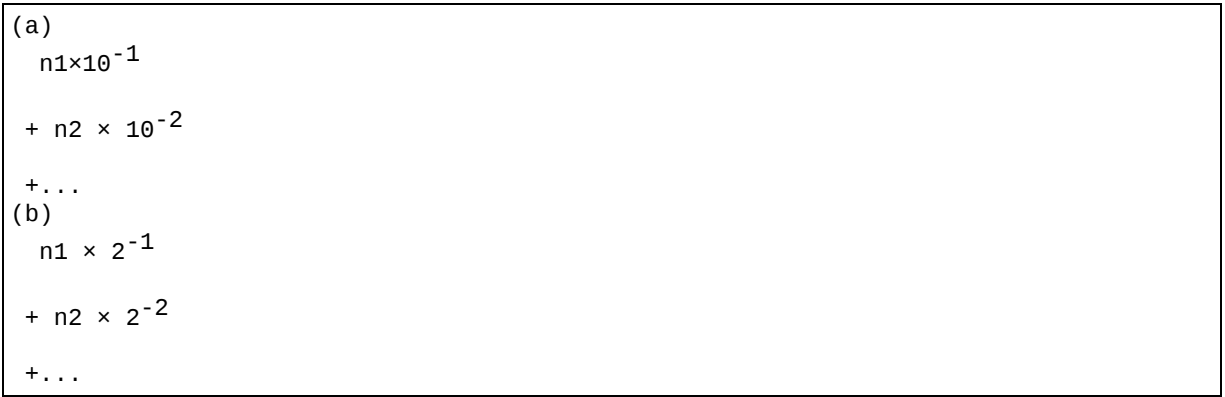


图 9-19 十进制与二进制的内部表示

比如，为了用二进制表示十进制小数 0.1 ，就写成 2 的幂（因为小于 1 ，所以幂是负数）相加的形式（参见图 9-20）。若一直持续下去，用二进制数表示的十进制数 0.1 就成了 $0.00011001100110011001100\dots$ 这种循环小数。在有效数字的范围内进行舍入，就会产生舍入误差。

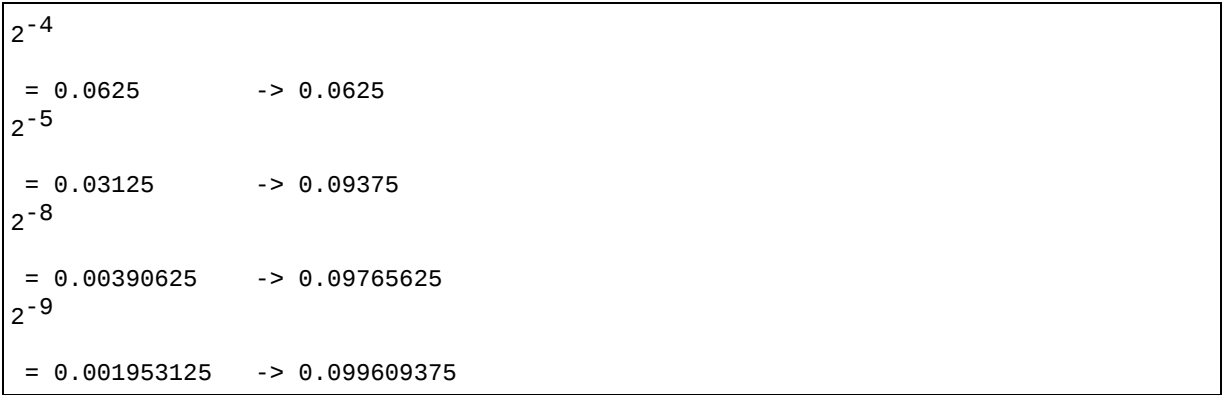


图 9-20 将十进制小数变为二进制的步骤

最大值溢出与最小值溢出

当运算结果超出了浮点数所能表示的数的范围时，就会发生最大值溢出或最小值溢出。超出最大值时称为最大值溢出，超出最小值时称为最小值溢出。

双精度的最大值定义为 **Float::MAX**，其值为 **1.79769313486232e+308**。如果硬要变成整数，就变成图 9-21 的样子。前 18 位以后是有效数字范围以外的数，没有意义。与双精度具有同样的 64 位宽度的整数所能表示的最大整数 **9223372036854775807** 相比，可以知道这个数要大得多。双精度所能表现的数的范围如此宽广，好像没问题，但大数之间的运算，却令人意外地很容易超出范围。如果运算结果超越这个范围，就成为无限大。

```
179769313486231570814527423731704356798070567525844996598917476803
157260780028538760589558632766878171540458953514382464234321326889
464182768467546703537516986049910576551282076245490090389328944075
868508455133942304583236903222948165808559332123348274797826204144
723168738177180919299881250404026184124858368
```

图 9-21 相当于 **Float::MAX** 的整数值，要比 64 位整数值大很多

还有一点必须注意，双精度所能表示的数的范围比整数大很多，随便将双精度变为整数，很容易就超出整数的范围。这可以说是另一种意义上的最大值溢出。

同样，双精度的最小值定义为 **Float::MIN**，其值为 **2.2250738585072e-308**。如果运算结果（的绝对值）比这个数值还小，就会发生最小值溢出，结果被置换为零。

非常大的数之间或非常小的数之间进行乘法运算的时候，发生最大值溢出或最小值溢出的可能性会增高。数值差别特别大的数之间也应避免乘法或除法运算。

信息丢失

与 64 位整数相比，双精度所能处理的数要大很多。但是，双精度之所以比 64 位信息全部处理的整数所能处理的范围大，是因为（双精度）数与数之间存在不能表示的数。特别是对于浮点数，数值越大，能够表示的“分解能”越低。在 0 附近，数值可以表示得很细致，离 0 越远，所能表示的数的间隔越大。

看看图 9-22。特别大的浮点数 **Float::MAX** 减去 1.0，结果与原来的数 **Float::MAX** 没有一点变化。

```
max = Float::MAX
max_minus_1 = max - 1.0

p max == max_minus_1 # => true (!)
```

图9-22 浮点小数的信息丢失从原来的数中减去1.0，大小不变

`Float` 的`==`方法返回 `true`，意味着 `max` 与 `max_minus_1` 在内部表示上完全一致。也就是说，不能因为要处理的数很大，就随随便便使用双精度数代替整数。数值大的时候，本来应该执行了 `increment`（值增加 1）操作，但结果并没有增加。两个浮点数相加的时候，两个数的指数部分要一致。计算 $1.0 \times 10^4 + 2.0 \times 10^2$ 的时候，首先要变成 $1.0 \times 10^4 + 0.02 \times 10^4$ 的形式，然后再计算结果。

```
1.02 × 104
= 10200
```

如果两个数的指数部分差别太大，指数部分变得一致时，较小数的尾数部分会变得太小，以至于不能表示，结果就会造成信息丢失。

为了避免信息丢失，需要在计算顺序方面想办法，最基本的是要避免特大数和特小数之间的运算。比如对很多很小的数进行合计，如果不动脑筋直接一个个相加的话，合计值会越来越大，它与每个加数之间的差别也会越来越大。也就是说，会发生信息丢失。对浮点数进行计算，比如要对分成很多组的某个要素进行合计时，需要注意信息丢失。

看一个具体例子吧。图 9-23 所示的程序要计算 4 个值的和：
10000001.0、0.12345678、0.11111111 和 -10000000.0。正确答案是 1.23456789，单纯计算时，结果会稍微有些不一样。

```
a = 10000001.0
b = 0.12345678
c = 0.11111111
d = -10000000.0

s = 0.0
s += a
```



```
s += b
s += c
s += d

p s # => 1.23456788994372
```

图 9-23 容易产生误差的合计程序，运算的顺序有问题

这是因为数值差别特别大的数 a 和 b ，在计算时发生了信息丢失。如果将计算顺序变成 $a \rightarrow d \rightarrow b \rightarrow c$ ，结果就变成 1.23456789，与正确值一致。由此可以看出，与乘法和除法一样，两个（绝对值）差别极大的数进行加法运算也会产生误差。

对数值差别很大的多个数进行合计时，如果像图 9-24 所示的那样处理，然后进行误差补正，就可以避免误差积累。

```
ar = [
  10000001.0,
  0.12345678,
  0.11111111,
  -10000000.0
]
s = r = 0.0;
ar.each do |x|
  t = s + x
  s = t + r
  r = t - s - x
end

p s+r # => 1.23456788994372
```

图 9-24 使误差变小的合计程序

不过，从图 9-24 的实际结果看，对 Ruby 所采用的双精度实数，重复次数少时，这种方法所能修正的误差还是有限的。这个结果很遗憾，修正前后的结果都完全一样。但请记住存在这么一种方法。

位数脱落

数值差别极大的两个数进行加法运算时，会发生信息丢失。数值几乎相同的两个数进行减法运算时，会发生位数脱落。数值相近的两个数

相减，结果与原来值相比非常小，有效位数会变少，这称为位数脱落。比如在图 9-25 的例子中，有效数字是 6 位的两个数相减，有效数字脱落为仅有 2 位。

$$\begin{array}{r} 1.23456 \times 10^{-2} \\ - 1.23444 \times 10^{-2} \\ \hline = 1.2 \times 10^{-6} \end{array}$$

图 9-25 发生位数脱落的运算例子

截止误差

浮点数的大小和精度都有限度，绝对无法表示除不尽的无理数。到某一位截止，总是有近似，这就产生了误差。

9.2.9 误差导致的严重问题

综上所述，浮点数有很多称为误差的陷阱。浮点数的误差可能导致比想象中更严重的问题。下面介绍两个事例⁶。

⁶ 这些事例是从 Francisco J. Santistive 先生的论文 *Robust Geometric Computation(RGC), State of the Art* 中引用的。翻译参考了 Radium Software Development

(<http://www.radiumsoftware.com/0506.html>)。

事例1

1991 年 2 月 25 日，海湾战争中美军的爱国者导弹拦截伊拉克军队的导弹失败，导弹命中美军营地，造成 28 名士兵死亡。失败原因在于，从导弹发射开始经过多少时间的计算有误差。

事例2

1996 年 6 月 4 日，欧洲宇航局（ESA）发射的无人阿丽亚娜 5 型火箭，发射后仅 40 秒就爆炸了。阿丽亚娜 5 型火箭的开发，花了近 10 年的时间，共耗资 70 亿美元。搭载在火箭上的器材总价值近 5 亿美元。事故的直接原因是，在惯性基准装置（IRS）内的软件中，将用

于表示水平方向速度的 64 位浮点数变换成了 16 位整数。这个数值超出了 16 位带符号整数所能表示的最大值 32768，造成变换失败。

这种严重事故虽然很少发生，但即使错误没那么严重，也请不要忘了，浮点数容易引起误差。

9.2.10 BigDecimal 是什么

浮点数运算的陷阱可以归结为两个原因：一是能够表示的精度有限，二是以二进制表示。

这些代价换来的是速度上的优势，所以需要权衡。但是，也有比起运算速度，更需要精度的时候。有不少语言，为了提高精度，提供了专用的类型或者类。

标准 Ruby 中提供了 **BigDecimal** 类，它有如下 3 个特征：

- 与 **Bignum** 一样，有效数字自动扩展；
- 以十进制计算；
- 以 C 语言记述，比内嵌的浮点数类（**Float**）要慢。

看一个使用 **BigDecimal** 的程序吧（参见图 9-26）。

```
require 'bigdecimal'
a=BigDecimal::new("0.123456789123456789")
b=BigDecimal("123456.78912345678",40)
c=a+b
puts c # => 0.123456912580245903456789E6
puts c+4 # => 0.123460912580245903456789E6
```

图 9-26 用 **BigDecimal** 运算的实例

使用 **BigDecimal**，需要加载 **bigdecimal** 库（第一行）。接下来指定字符串生成 **BigDecimal** 对象（第 2 行，第 3 行）。第 3 行的第 2 个参数指定精度（有效数字的十进制位数）。

运算正常进行（第4行）。即使有整数等其他类型的数混在其中，也会进行适当变换后再运算（第6行）。

最初设计时，会觉得怎么是以字符串的形式给出初始值呢？仔细想想，几乎所有的情况，浮点数都是以十进制的字符串读进来的（数据文件等），因此非常合理。

9.2.11 能够表示分数的Rational 类

除不尽的数中，有很多能够以分数表示的数（有理数）。能够直接表示为分数的是 **Rational** 有理数）类（参见图 9-27）。

```
puts 1.quo(4)    # => 0.25
require "rational"
a = Rational(1,3) # => 正确的1/3
puts a*3         # => 有理数1/1
p a*3
puts 1.quo(4)    # => 有理数1/4
```

图 9-27 使用 Rational 类的例子

图 9-27 的 `quo` 是除法运算的方法，返回系统设置下能够得到的最佳结果。`quo` 方法的运算结果，在加载 **Rational** 类时是 **Rational**，未加载 **Rational** 类时是 **Float**。

现在，与 **BigDecimal** 一样，**Rational** 类位于标准库中，而使其像 **Bignum** 那样成为内嵌类的尝试也在进行中。

没有常识的计算机

从孩提时使用计算器的时候，就很惊异于这样的事实：1 除以 3 得到的数，连加 3 次却不是 1。

知道了 1 不能被 3 除尽，也就理解了从某种程度上讲这也是没办法的事。但是，1 除以 10 得到的能够除尽的数，连加 10 次却不是 1，这样的事不管怎么说都是违反常识的。

虽然计算机在一定程度上反映了现实世界，但是实际上它所提供的顶多只是“幻影”，经常会与现实世界中人的思考发生偏差。

计算机的浮点数就是特别容易违反常识的领域。内部以二进制表示数（0.1 在二进制中是除不尽的，10 个 0.1 相加不能正好得到 1），浮点数计算中有误差，等等，很多地方应当注意。结果就成了计算机有计算机的常识。为了获得计算效率，某种程度上也是没办法的事。

但是，作为高级程序员，应该有更高的目标。如果计算机中能够自然计算的整数有上限，就要想办法引入多倍长以超越界限；为了能除尽，就引入有理数等。如何权衡计算效率以达到最大限度的平衡技巧，我想，这种技巧正是区分普通程序员与一流程序员的一条界线吧。

第 10 章 高速执行和并行处理

10.1 让程序高速执行（前篇）

计算机的处理速度正在以惊人的气势提高着。我们现在使用的计算机，比过去的超级计算机的性能还要优良，而与半世纪前问世的计算机相比，性能提升了数十万倍。

尽管计算机已如此高速，但是人的欲望却没有止境。对于程序员来说，程序的执行速度像是永久的课题。让程序高速执行，有时甚至会让人觉得，“那么慌张是要去哪儿？”

下面，讲解一下关于高速执行的“秘密”、“界限”以及“战略”。

10.1.1 是不是越快越好

考虑程序高速执行（性能优化）之际，要先仔细想想。程序高速执行并不是一直所期望的。与其他各种各样的因素一样，性能也要权衡利

弊。如果总是视速度最重要，那么就总得准备最高速的机器。或许，还需要用能够编写最高速程序的低级语言（比如汇编）来写程序。

但是，并不是视速度最优先就一定好。预算、开发效率和开发周期等制约因素也都在性能权衡范围之内，在提高速度方面所付出的代价，应该只是值得的那么多。

比如用 Ruby 写一个程序，处理 100MB 的数据。写程序花了 30 分钟，执行花了 2 小时，加起来是 2 小时 30 分。同样的程序想要在 30 分钟内执行完，用 C 语言写花了 8 小时，结果怎样？执行时间是变短了，但加起来要 8 小时 30 分。哪一个合算就不用说了。

但如果这个程序每天都重复执行，每天都要耗费 2 小时等结果，这就完全不一样了，用 C 语言花 8 小时来开发就更值得了。

啰里啰嗦再说一遍，性能需要权衡。通常，程序能在必要的时间内执行完毕就足够了。我不认为，一味追求高速而什么制约因素都不考虑有多么聪明。

10.1.2 高速执行的乐趣与效率

对程序员来说，让程序高速执行本身是一种智力上的挑战。找出慢在什么地方，推测并改善问题点，程序执行就会变快。就像解开某种谜团一样，有种乐趣。改进的结果，可以体现在明确的执行时间上。也许可以说，性能优化的成就感在编程中让人觉得最充实。

这倒不是什么坏事，但工作中光有成就感还是不够的。假设我的笔记本电脑价值 20 万日元，可以用 3 年，换算一下，平均每秒仅仅 0.00211 日元。我的一小时工资假设是 760 日元（这比实际值要低很多）。为了让程序执行快 10 秒，我花了一小时修改程序。要赚回这一小时的钱，那个程序非得经常重复执行才行。简单计算一下，要执行 36 000 次才能赚回 760 日元。

暂且不谈兴趣编程，工作中在进行性能优化之前，必须确认是否真有必要提高速度。速度提高到什么程度也要事先估算。

10.1.3 以数据为基础作出判断

与普遍的看法不同，编程是文科因素占很大比重的领域。而性能优化却切切实实属于理科领域。性能优化首先要考虑客观性。不要仅仅抱怨慢，而要测定，用数据来说话。

标准 Linux 中有测定执行时间的工具，就是 **time** 命令。其用法很简单，命令执行时只要在开始加上 **time** 就行了（参见图 10-1）。图 10-1 的格式是嵌入 **bash** 的 **time** 命令。除此以外，**time** 命令还有多个版本，格式有所不同，但即使如此，大多都包含以下 3 种数值。

```
% time ruby sample/fact.rb 20
2432902008176640000

real    0m0.005s
user    0m0.005s
sys     0m0.001s
```

图 10-1 **time** 命令的使用。计算 20 的阶乘。使用的程序是 Ruby 附带的例程 **fact.rb**

- **real**（总执行时间）。程序从开始到终止的执行时间。有时称为 **total** 或者 **elapsed**（经过时间）。
- **user**（用户消费时间）。程序执行中，用户所消费的时间，即程序本身的执行时间。
- **sys**（系统消费时间）。程序执行中，系统调用（**system call**）所花费的时间，有时称为 **system**。

程序执行得快慢，虽然可以用 **real** 来判断，但通过观察 **user** 与 **sys** 的比率，可以大体判断出是程序本身执行慢，还是系统调用太多而导致了程序变慢。

也许你没怎么意识到，系统调用花费的代价（时间）很多。通常，程序工作的用户空间与系统调用所工作的内核空间是完全隔离的，所以系统调用需要遵循以下几个步骤：（1）将参数从用户空间复制到内核空间；（2）执行系统调用的中断程序；（3）在中断处理内切换到内核空间。将系统调用的结果返回用户空间又需要反方向执行这几步。所以，如果有太多系统调用，就会引起性能恶化。

10.1.4 改善系统调用

现在就来看看减少系统调用的次数，性能能够改善到什么程度吧。

图 10-2 所示是将当前目录中的文件名按照文件更新时间进行排序的程序。这个程序在 500MHz 的 PentiumIII 上运行。排序对象是有 4556 个文件的目录（ruby-core 的邮件履历）。结果如下。

```
Dir.entries(".").sort{|a,b|  
  File.mtime(a) <=>  
  File.mtime(b)  
}
```

图 10-2 改进前程序

```
real    0m4.624s  
user    0m3.590s  
sys     0m0.850s
```

程序执行用了 4.6 秒。系统调用的时间为 0.850 秒，约占总时间的 20%。如果减少系统调用会有多大效果呢？

首先，统计一下图 10-2 所示程序中的 `File.mtime()` 调用了多少次。在 `sort` 方法中，如果给出程序块 `{...}`，为了排序，每次比较元素时都要调用此程序块。所以，程序块的调用次数比元素个数要多很多。稍微调整一下上述程序，数一数程序块被调用了多少次。结果发现，比较元素时程序块执行了 78 270 次。为了比较 4556 个元素，就调用了程序块 78 270 次，太频繁了。每次执行此程序块，都要调用两次 `File.mtime()`，总体上要调用 156 540 次。

排序处理任务重的时候，典型的对策是使用施瓦茨变换（Schwarzian Transform）。这是 Randal Schwarz¹ 设计的高速排序法，先计算出比较用的值，避免比较计算重复进行。施瓦茨变换按以下步骤进行。

¹ Randal Schwarz 也是 *Programming Perl* 第 1 版及第 2 版的合著者。

1. 首先，对于排序对象的各个元素，计算比较用值，与元素本身配对，组成一个数组（每个数组元素是一个二元数组）。
2. 将此数组的数组按照以上计算的比较用值进行比较。
3. 从排序后的数组的数组中取出以前的元素。

上述步骤用程序来表示如图 10-3 所示。稍微有点烦琐。事实上，在 Ruby 中，施瓦茨变换内嵌在了 `sort_by` 方法中。用 `sort_by`，可以不考虑复杂的事情而直接使用施瓦茨变换。使用 `sort_by` 的程序如图 10-4 所示。

```
Dir.entries(".").
  map{|x| [x,File.mtime(x)]}.
  sort{|a,b| a[1] <=> b[1]}.
  map{|x| x[0]}
```

图 10-3 使用施瓦茨变换的排序程序

```
Dir.entries(".").sort_by{|a|
  File.mtime(a)
}
```

图 10-4 用 `sort_by` 方法的排序程序

与最初的程序相比，既变得短小精悍，又将“使用此值进行比较”的意图明确地表达出来。快点执行一下看看。其结果如下。

```
real    0m0.263s
user    0m0.220s
sys     0m0.050s
```

总执行时间从 4.624 秒减少为 0.263 秒，用户消费时间从 3.590 秒减少为 0.220 秒，系统时间从 0.850 秒减少为 0.050 秒，大体上快了 17 倍。用其他方法测定 `File.mtime()` 的调用次数为 4556 次，大约减少为最初的 1/34。

程序变得更短更易懂，而且执行速度快了 17 倍，如愿以偿。

施瓦茨变换通过事先保存反复计算的值来削减了计算量。这是一种称为 **Memoize** 的高速技术。在计算中，时间与空间通常可以交换。保存计算结果要占用多余的内存，作为回报，能够节省时间。**Memoize** 技术在各种情况下都可以用于提高速度。

10.1.5 数据可靠吗

或许这次只是偶尔做得好。关于 **time** 的测定结果，好像需要再多讲一点。

首先重要的是，像 **Linux** 这种多任务操作系统，**CPU** 始终用于执行多个进程。所以，用 **time** 所测的总执行时间会受到其他进程的影响。前面的例子也如此，用户消费时间加上系统消费时间与总执行时间不一致。

还有一点，就是误差。**bash** 的 **time** 命令表示到小数点后 3 位。考虑到其他进程的影响，操作系统的时钟性能等因素，对于非实时操作系统的 **Linux**，时间达不到小数点后 3 位（1 毫秒），顶多也就小数点后 2 位（1/100 秒）。

而根据操作系统的缓存与换页的不同，虽执行同样的命令，但执行时间有可能极为不同。

考虑到这些情况，在测定执行速度时，有必要注意以下条件。

- 避免测定太短的时间。
- 反复测定。

测定太短的时间（比如 1 秒以下），误差太大，得不到有意义的结果。另外，为了尽可能排除其他进程以及缓存与换页的影响，反复测定要达一定的次数，去掉测定结果中性能极差的几个，观察总体倾向。

10.1.6 只需改善瓶颈

性能优化中，“因为是排序，所以就用施瓦茨变换”这种条件反射式的对策并非总管用。大到一定程度的程序，问题是不是真的在于排序部分，判断起来并不容易。为了合理提高速度，确立恰当的策略是很必要的。

为此，首先必须理解帕雷托法则。帕雷托法则又称 80/20 法则，即 80%的数值是由 20%的构成要素产生的。19 世纪后半叶，由意大利经济学家 Vilfredo Federico Damaso Pareto 发现而得名。由帕雷托法则可知，有 20%的努力可以得到巨大回报，而有 80%的努力得不到多少回报。在得不到回报的地方，不管怎么努力都是徒劳的。

Donald Knuth² 也提到，通常一半以上的执行时间都耗费在程序中不到 4%的部分。

² Donald Knuth 是计算机科学家。他开发了 TeX，撰写了多卷本《计算机程序设计艺术》。

这些耗费了大半以上执行时间的部分称为瓶颈。对瓶颈以外的部分，不管倾注多少劳力，都是在浪费。对于只耗费总执行时间 1%的部分，费了半天功夫，即使处理时间缩短了 50%，总执行时间也只是缩短了 0.5%。0.5%的速度改善，恐怕还抵不上测定误差。反过来，耗费 80% 执行时间的部分，就算执行速度只提高 20%，总执行时间就会提高 16%。

改善瓶颈在性能优化中是最最基本的。不先确认瓶颈就进行高速化处理是不行的。

确认一下刚才程序的瓶颈在哪儿吧。判定瓶颈，可以用 profiler 这一工具。Ruby 在执行时，解释器中附加了 -rprofile 选项，就可以利用 profiler 了。假设图 10-2 所示的改进前程序保存在文件 sort1.rb 中，像下面这样执行。

```
% ruby -rprofile sort1.rb
```

程序的执行结果请参见图 10-5，各栏的含义 请参照表 10-1。

%	cumulative	self	self	total
---	------------	------	------	-------

time	seconds	seconds	calls	ms/call	ms/call	name
41.10	47.59	47.59	156540	0.30	0.40	
File#mtime						
37.85	91.42	43.83	1	43830.00	115780.00	
Array#sort						
12.92	106.38	14.96	156541	0.10	0.10	
Kernel.respond_to?						
8.12	115.78	9.40	78270	0.12	0.12	Time#
<=>						
0.03	115.81	0.03	1	30.00	30.00	
Profiler_.start_profile						
0.01	115.82	0.01	1	10.00	10.00	
Dir#each						
0.00	115.82	0.00	1	0.00	10.00	
Dir#entries						
0.00	115.82	0.00	1	0.00	0.00	
Dir#open						
0.00	115.82	0.00	1	0.00	10.00	
Enumerable.to_a						
0.00	115.82	0.00	1	0.00	115790.00	
#toplevel						

图 10-5 profiler 的执行结果，图 10-2 所示程序改进前的测定结果

表10-1 图10-5和图10-6中profiler结果的解释

栏 (列)	含 义
1	执行时间的比例 (%)
2	累计执行时间 (秒)
3	各方法的执行时间 (秒)
4	调用次数 (次)
5	每次调用该方法所消费的时间 (毫秒)
6	每次调用所消费的时间 (毫秒)
7	方法名

第 5 栏和第 6 栏或许有些难懂。第 5 栏不包含被调用的方法所花费的时间，而第 6 栏包含这个时间。

从图中的结果来看，**File.mtime()** 被调用 156 540 次，耗费了总执行时间的 41.1%。这个方法就是瓶颈。

那么，使用 `sort_by` 会怎样呢？对于图 10-4 中使用 `sort_by` 的程序，`profiler` 的执行结果如图 10-6 所示。

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
56.28	14.30	14.30	1	14300.00	25400.00	
Enumerable.sort_by						
31.64	22.34	8.04	78270	0.10	0.10	Time#<=>
5.08	23.63	1.29	4556	0.28	0.40	File#mtime
4.84	24.86	1.23	1	1230.00	3060.00	Array#each
2.13	25.40	0.54	4557	0.12	0.12	
Kernel.respond_to?						
0.16	25.44	0.04	1	40.00	40.00	
Profiler.start_profile						
0.04	25.45	0.01	1	10.00	10.00	Dir#each
0.00	25.45	0.00	1	0.00	10.00	
Dir#entries						
0.00	25.45	0.00	1	0.00	0.00	Dir#open
0.00	25.45	0.00	1	0.00	25410.00	#toplevel
0.00	25.45	0.00	1	0.00	10.00	
Enumerable.to_a						

图 10-6 `profiler` 的执行结果，图 10-4 所示程序改进后的测定结果

10.1.7 `profiler` 本身成了累赘

仔细看看图 10-5 和图 10-6 就可以知道，使用 `profiler` 使得程序执行时间大幅延长。不使用 `profiler` 执行时，改进前是 4.624 秒，改进后是 0.263 秒。而使用 `profiler` 执行时，改进前是 115.82 秒，改进后是 25.45 秒。前者执行时间延长至原来的 25 倍，后者执行时间则是原来的 97 倍。

从 `profiler` 所引起的执行速度降低来看，不禁让人想起量子力学的不确定性原理。不确定性原理是指测定行为本身对被测对象产生了影响，从原理上可以说不可能正确知道对象的状态。应当认识到，从 `profiler` 的测定结果来看，只能知道实际状态的大体倾向。

10.1.8 算法与数据结构

在进行性能优化时，按下述步骤去改善。先找出瓶颈，明确知道应当改善哪里后，再考虑算法和数据结构的选择。去除多余的赋值，把反复计算的值放入变量中，这些都是小技巧，性能顶多改善几成。但是，若能选用一个好的算法，有时会改善数十倍，甚至上百倍。

选择合适的算法，这是性能改善的第一考虑因素。要记住这条铁则。

10.1.9 理解O 记法

如何从效率方面判定一个算法的好坏呢？一个方法就是 O 记法。O 记法表示某种算法对于变量（比如使用的元素个数）如何变化。

比如基本排序算法的冒泡排序算法，所有元素要全部比较。也就是比较次数与元素个数（通常以 n 来表示）的平方成比例。所以，这种算法的计算量是 $O(n^2)$ 。各种算法的计算量按效率高低顺序总结在表 10-2 中。

表10-2 各种算法的计算量

$O(1)$	数组访问·哈希法
$O(\log(n))$	binary search
$O(n)$	字符串比较
$O(n \cdot \log(n))$	quick sort
$O(n^2)$	冒泡排序
$O(n^3)$	行列乘法运算
$O(2^n)$	集合分割问题

虽说 O 记法相同，但两种算法应用于同一数据，结果不一定就相同。有时会出现明显的性能差异。

假设某一算法的计算量对元素个数来说是 n^2 ，另一算法的计算量是 $n^2 + 3n$ ，随着数据量的增加，两种算法的表现不同。但是，O 记法中最大项（这里是 n^2 ）以外的项被忽视，这两个算法都写成 $O(n^2)$ 。

请注意 O 记法只能表示随着元素个数的增加，该算法的大体走向。

O 记法 $O(n^2)$ ，表示数据量变成 10 倍的时候，执行时间按元素个数的 2 次方比例增长这一趋势。并不意味着处理 100 件花了 1 秒，处理 1000 件就一定花 100 秒。

10.1.10 选择算法

现在来看看算法的选择对于性能的影响。图 10-7 中的程序是从给出的两个数组中，求出两方都含有的元素。

```
def intersect1(a, b)
  result = []
  for x in a
    for y in b
      result.push(x) if x == y
    end
  end
  return result
end
```

图 10-7 求出给定两个数组中共同含有的元素的程序，计算量为 $O(n \cdot m)$

图 10-7 中，使用二重循环对两个数组的元素全部进行比较。该算法的变量是作为参数传过来的两个数组的长度。假设长度为 m 、 n ，则该算法的 O 记法为 $O(n \cdot m)$ 。随着各自的数组变大，必要的计算量与两个数组的长度的积成比例。

想要改善这个算法的计算量，只能通过改变数据结构。图 10-8 使用哈希表，计算量才得以改善。

```
def intersect2(a, b)
  hash = {}
  result = []
  # 准备哈希表
  for x in a
    hash[x] = true
  end
  # 哈希表中有的话，就追加
  for y in b
    result.push(y) if hash.key?(y)
  end
  return result
end
```

```
end
```

图 10-8 求出给定的两个数组中共同含有的元素的程序，使用哈希表，计算量限制在 $O(n+m)$

图 10-8 的程序中，首先用数组元素构造了一个哈希表。哈希表这种数据结构，其元素的存在检查只花费常数时间，计算量是 $O(1)$ 。利用这一点，可以将计算量削减为 $O(m+n)$ 。为了哈希表的实现和检查，需要对两个数组的各个元素进行循环，这对计算量有影响。相对于图 10-7 中的程序，在数组变得很大的时候，图 10-8 中的程序所需的处理时间要短得多。

10.1.11 调查算法的性能

实际比较一下两种算法的性能。可以准备两个独立的程序，然后用 `time` 命令。这里使用 Ruby 提供的 `benchmark`（基准）。图 10-9 是在进行算法性能比较时用的 `benchmark` 程序。

```
require 'benchmark'

a = (1..10000).collect{rand(1000)}
b = (1..10000).collect{rand(1000)}

Benchmark.bm do |x|
  x.report { intersect1(a,b) }
  x.report { intersect2(a,b) }
end
```

图 10-9 `benchmark` 程序（基准程序）使用 Ruby 标准的 `benchmark` 库

实际上，图 10-9 中的程序要加上图 10-7 和图 10-8 中程序的方法定义。程序执行结果如图 10-10 所示³。

3 是在以 1.6GHz 运行的 PentiumM，786M 内存的 Thinkpad X31 上的测试结果。OS 是 Linux 2.6。

user	system	total	real
38.430000	0.100000	38.530000	(45.371555)
0.010000	0.000000	0.010000	(0.011314)

图 10-10 benchmark 的执行结果。图 10-7 与图 10-8 中程序的调查结果

$O(n \cdot m)$ 的算法需要花费 38.5 秒来处理，与此相对， $O(n + m)$ 的算法只需要花费 0.01 秒。实际处理时间缩短至 $1/3850$ 。三千倍以上的性能改善，如果不靠改变算法，是不可想象的。

10.1.12 高速执行的悲哀

回顾编程历史，高速执行引起了各种各样的悲喜剧。高速执行的基准是程序的执行速度，但执行速度依赖于各种各样的因素，若随随便便就进行性能优化，就可能会落入陷阱。就连这次举出的简单的例题，也有多个陷阱。

徒劳无益的努力

比如，在高速执行上花费过多的时间，不知不觉中，很容易在跟瓶颈无关的地方花费太多徒劳无益的努力。有时候，得到数千倍性能改善的这种成就感，对于程序员而言，起到了一种麻药似的作用。

改良绊住了手脚

`sort_by` 所依据的施瓦茨变换，是用时间和空间的交换来削减计算量的方法。`sort_by` 方法与 `sort` 方法相比，占用了多达 3 倍以上的内存。所以，模块内的处理本来不是很重，如果一味地占用内存反而可能会变慢。

性能优化，光有理论还不管用。手头上有许多实验结果，至少现在从 Ruby 上的结果来看，排序算法要想超过 `sort_by` 是很困难的。我想大家尽可以放心活用 `sort_by`，直到 `profiler` 证明 `sort_by` 是瓶颈。

像这样，如果不实际做的话就不知道到底什么是正确的，这也可以说是高速执行的陷阱之一。

算法选择的圈套

刚才，为了取得两个数组中共同的元素，以图10-8中使用哈希表的程序代替了图10-7中使用二重循环的程序，实现了多达3850倍的提速。一般来说，到此也就满足了，但仔细检查一下就会发现，在数组类 **Array** 中，已经提供有一个求共同元素的内嵌方法 **&**。Ruby 标准库的方法，很多都是用 C 语言实现的，使用 **&**，估计执行会更快吧。实际运行一下 **benchmark**，在同样条件下，执行时间是0.001529秒。执行速度是图10-8中程序的7倍。像这样，即便觉得已经找到了最好的算法，也可能还存在更好的实现方法。千万不可粗心大意。

一般地，尽可能活用 Ruby 原装的内嵌方法，这是 Ruby 中实现高速执行的窍门。Ruby 的内嵌方法除了算法精炼之外，几乎都是以 C 实现的，因此，高速执行的可能性很大。

但是，还有后话。在 **benchmark** 程序中，已经知道，图10-8中的程序比图10-7的快，**&**方法比图10-8中的快。但这3个程序的执行，严格来说并不一样。在数组元素重复的时候，执行也是不一样的。

在进行性能优化时，不改变原来程序的执行是一个大原则，像这样在特殊情况下执行上的差异，需要考虑其对程序的目的有多大影响。

类似这样的情况，数组元素的重复有没有可能，或者是元素重复时执行上的差异能否接受，都必须要充分考虑到。不过，既然有几万倍（速度上）的差异，稍微有点（执行上）的差异，还是可以接受的。

10.1.13 性能优化的格言

最后，介绍几条关于性能优化的格言。第 1 条也是最有名的。

过早的优化是万恶之源。

还有一句也说的是这个意思。

优化有两条准则。

- 别做优化。
- （仅适用于专家）先不要做优化。

这算是对于那些热衷于做高速优化的程序员的警戒格言吧。

10.2 让程序高速执行（后篇）

本节作为程序高速优化的实践篇，对具有一定规模的具体程序，一边进行阶段性的高速优化，一边学习实践技术。

例题是曼德勃罗集合¹的计算程序（参见图 10-11）。曼德勃罗集合是非常美的集合，本来应该用图像表示²。但使用 GUI 以后，需要大量依赖于特定 GUI 库（Graphics Routine）的代码。因为这次的主题是高速执行，没有直接关系的 GUI 代码尽可能要省去，所以用字符（英文字母）来表现图形。

1 曼德勃罗集合是指复平面上满足以下条件的复素数的集合：经过某种反复迭代运算以后，其值不会发散到无限大。

2 用 GUI 编写的曼德勃罗集合描述程序，在《面向对象编程语言 Ruby》（Ascii 出版社，ISBN 4756132545）的第 9 章中有讲述。但是，现在广泛使用的 Ruby/Gtk2 中，由于 API 变了，因此不能再运行。

改为以字符输出，结果减少了整体计算量，带来缩短测定时间的附加效果。进行高速优化需要频繁测定，缩短每次的执行时间是一个优点。

依存于特定 API 的 GUI 程序，会随着 API 的变更而变得不再好用，但以字符为基础的程序能够长期放心使用。

10.2.1 确认程序概要

图 10-11 是计算曼德勃罗集合的算法代码，比 GUI 版要简单³。GUI 版的代码有 102 行。

3 图 10-11 中所列举的曼德勃罗集合计算程序是基于 Steven N. Severinghaus 的程序（<http://severinghaus.org/projects/mandelbrot>）。

```
1 require 'complex'
2
```

```

3 def mandelbrot(cr, ci)
4   limit=95
5   iterations=0
6   c=Complex.new(cr,ci)
7   z=Complex.new(0,0)
8   while iterations<limit and z.abs<10
9     z=z*z+c
10    iterations+=1
11  end
12 return iterations
13 end
14
15 def mandel_calc(min_r, min_i, max_r, max_i, res)
16   cur_i = min_i
17   while cur_i > max_i
18     print "|"
19     cur_r = min_r
20     while cur_r < max_r
21       ch = 127 - mandelbrot(cur_r, cur_i)
22       printf "%c",ch
23       cur_r += res
24     end
25     print "|\n"
26     cur_i -= res
27   end
28 end
29
30 mandel_calc(-2, 1, 1, -1, 0.04)

```

图 10-11 曼德勃罗集合的计算程序，文件名为 `mandel1.rb`。未进行任何优化

程序结构很简单。第 15 行到第 28 行的 `mandel_calc` 方法计算指定范围的曼德勃罗集合。这次为了看起来美观，在第 30 行指定了范围（坐标） $(-2, 1) - (1, -1)$ ，第 5 个参数是分辨率。一格（一个字符）相当于 0.04。当这个值变大时，结果就会变小。

内部调用 `mandel_calc` 方法的，是第 3 行到第 13 行的 `mandelbrot` 方法。只计算某一点的状态。

先简单执行一下看看。

```
% ruby mandel1.rb
```


性能优化，首先找出瓶颈（问题所在）是一条铁则。如前所述，对于发现瓶颈，**profiler** 是一个有用的工具。使用 **profiler** 测定一下吧。

```
% time ruby -r profile mandel1.rb
.....
real    4m4.872s
user    3m36.026s
sys     0m4.428s
```

profiler 的结果如图 10-13 所示。从 **profiler** 的结果来看，耗费时间（self seconds）最多的 3 个方法如下所示（括号内是调用次数）。

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
20.85	45.05	45.05	229539	0.20	0.57	Object#Complex
19.22	86.56	41.51	237039	0.18	0.23	Complex#initialize
11.20	110.75	24.19	114769	0.21	0.89	Complex#*
7.68	127.34	16.59	3750	4.42	57.29	Object#mandelbrot
7.22	142.93	15.59	1181444	0.01	0.01	Kernel.kind_of?
6.88	157.79	14.86	114769	0.13	0.73	Complex#+
4.39	167.27	9.48	459078	0.02	0.02	Numeric#imag
4.31	176.58	9.31	459078	0.02	0.02	Numeric#real
3.28	183.67	7.09	237039	0.03	0.26	Class#new
2.40	188.86	5.19	443822	0.01	0.01	Float#*
2.12	193.44	4.58	117551	0.04	0.05	Complex#abs
2.04	197.85	4.41	355614	0.01	0.01	Fixnum#+
2.00	202.16	4.31	336751	0.01	0.01	Float#-
1.97	206.41	4.25	336751	0.01	0.01	Float#+
1.47	209.58	3.17	225709	0.01	0.01	Float#==
0.85	211.41	1.83	117551	0.02	0.02	Math.hypot
0.74	213.01	1.60	121301	0.01	0.01	Float#<
0.67	214.46	1.45	118569	0.01	0.01	Fixnum#<
0.25	215.01	0.55	1	550.00	216020.00	Object#mandel_calc
0.19	215.43	0.42	3750	0.11	0.14	Kernel.printf
0.12	215.69	0.26	15254	0.02	0.02	Fixnum#*
0.07	215.85	0.16	11357	0.01	0.01	Fixnum#-
0.04	215.94	0.09	3850	0.02	0.02	IO#write

0.04	216.02	0.08	3830	0.02	0.02	Fixnum#==
0.00	216.02	0.00	37	0.00	0.00	
Kernel.singleton_method_added						
0.00	216.02	0.00	34	0.00	0.00	
Module#module_function						
0.00	216.02	0.00	1	0.00	0.00	
Module#method_undefined						
0.00	216.02	0.00	1	0.00	0.00	
Class#inherited						
0.00	216.02	0.00	2	0.00	0.00	Module#attr
0.00	216.02	0.00	1	0.00	0.00	
Kernel.require						
0.00	216.02	0.00	1	0.00	0.00	Fixnum#>
0.00	216.02	0.00	71	0.00	0.00	
Module#method_added						
0.00	216.02	0.00	50	0.00	0.00	Float#>
0.00	216.02	0.00	100	0.00	0.00	Kernel.print
0.00	216.02	0.00	1	0.00	216020.00	#toplevel

图 10-13 图 10-11 中代码的 profiler 结果

(1 位) Object#Complex (229 539 次)

(2 位) Complex#initialize (237 039 次)

(3 位) Complex#* (114 769 次)

`Complex()` 是复素数生成方法，`Complex#initialize` 是复素数的初始化方法。这些都是被 `mandelbrot` 方法调用的（而且 `mandelbrot` 方法本身也是第 4 位），`mandelbrot` 方法就是瓶颈这一点应该不会错。因此，高速优化就以 `mandelbrot` 方法为对象吧。

10.2.3 使用更好的 profiler

话题稍微岔开一下，对 `profiler` 本身的执行时间有些在意。不是慢一点半点。`Ruby` 的 `profiler` 是以 `Ruby` 本身来写的，有人说不怎么快。事实上大家都觉得好像太慢了。

原始的曼德勃罗集合计算程序，在不使用 `profiler` 执行时，需要 2.6 秒；在使用 `profiler` 执行时，需要 4 分零 5 秒。执行时间变成了不使用 `profiler` 执行时的 94 倍。

实际上，虽然不是标准提供，但是有更好的 **profiler**，就是称为 **ruby-prof** 的程序⁴。通过使用扩展库，可以实现高速 **profile**。

ruby-prof 可以从以下的 URL 得到 (<http://rubyforge.org/projects/ruby-prof>)。开发者是前田修武。

那么，来试试 **ruby-prof** 吧。为了用 **ruby-prof** 进行 **profile**，用 **ruby-prof** 命令取代 **ruby** 命令来执行。**time** 命令就不需要了。

```
% ruby-prof mandel11.rb
.....
real      0m12.047s
user      0m6.336s
sys       0m4.004s
```

ruby-prof 有很多优点，其中最重要的是它能够高速执行，仅仅花 12 秒。跟什么都不做（2.6 秒，不用任何 **profiler**）比起来虽然有些慢，但跟标准 **profiler** 的 4 分多钟比起来，还是要快得多。

10.2.4 高速优化之一：削减对象

知道了问题所在，下一步是程序的高速化。从刚才 **profiler** 的结果来看，**mandelbrot** 方法占用了多半的执行时间，而且复素数的计算成为瓶颈。这里，就考虑除去瓶颈——复素数的计算。复素数以实数和虚数和的形式来表示。所以，需要特别的运算。

mandelbrot 方法的核心是复素数的计算 $z = z * z + c$ 。复素数的加法、乘法和绝对值 **abs** 定义于图 10-14。使用这个算式，如果能独立计算实部和虚部的话，那么不用 **Complex** 类也行。

```
class Complex
  def +(other)
    return Complex.new(@re+other.re,@im+other.im)
  end
  def *(other)
    return Complex.new(@re*other.re-
@im*other.im,@re*other.im+@im*other.re)
  end
```



```

def abs
  return Math.sqrt(@re**2+@im**2)
end
end

```

图 10-14 复素数类的加法、乘法和绝对值的定义内容

避开了 **Complex** 类的生成，最花时间的 3 个方法就可以削减了，执行速度的改善应该是有希望的。

图 10-15 所示的是根据这个方针改善后的 **mandelbrot** 方法。

```

def mandelbrot(cr, ci)
  limit=95
  iterations=0
  zr = zi = 0.0
  while iterations<limit and Math.sqrt(zr**2+zi**2)<10
    zr, zi = zr*zr-zi*zi+cr, zr*zi+zi*zr+ci
    iterations+=1
  end
  return iterations
end

```

图 10-15 mandelbrot 方法（改善版 1）

最初的 **mandel1.rb** 的 **mandelbrot** 方法被置换以后，请以文件名 **mandel2.rb** 来保存。因为 **mandel2.rb** 不含复素数计算，“**require 'complex'**”这一行可以删除。

执行一下 **mandel2.rb** 吧。

```

% time ruby mandel2.rb > /dev/null
real    0m0.517s
user    0m0.512s
sys     0m0.000s

```

执行时间变成了 0.5 秒。与之前的版本相比，快了 5 倍之多，真了不起！

那么，再回过头来考察一下 `mandel2.rb` 执行时间变短的原因吧。首先，最大的原因是不再使用复素数。这样也就避免了复素数的生成所耗费的时间。实际上占用执行时间 40% 的部分被削减掉了。

由此可以得出以下的 Ruby 高速优化的规则 1。

规则1：减少对象

使用高级面向对象语言却不得使用对象，这条件太苛刻了。但对象的生成要花费一定的成本（时间），对于那些频繁地生成以至于影响到执行速度的对象，有时需要采取些对策。`mandelbrot` 程序中，需要大量生成复素数，以至于影响到执行速度，恰好是需要采取措施的地方。

减少对象，除了会降低生成成本以外，还有别的好处。

Ruby 中不再使用的对象，由被称作垃圾收集（`garbage collection`）的处理自动进行回收。垃圾收集处理需要检查对象的引用关系，任何地方都不再引用的对象会被判定为已经不再使用了。所以，当对象的数量很多的时候，这种“是不是已经不再使用了”的判断成本就要增大。`profiler` 不检查垃圾收集处理，这样就容易看漏，当怀疑对象是不是太多了的时候，有必要检查一下。

但仅仅是不再生成复素数，只能达成 40% 的时间削减，并不能说明 `mandel2.rb` 为何快了 5 倍。

图 10-16 显示的是速度改善后的 `mandel2.rb` 的 `profile` 结果。看了这个图，首先注意到的是，行数少了很多。原始的 `mandel1.rb` 有 35 行，与此相对，`mandel2.rb` 只有 18 行。这是由于避开了 `complex` 库，调用方法（的种类）也变少了的原因。

% time	cumulative seconds	self seconds	calls	self ms/call	total ms/call	name
62.42	31.83	31.83	3750	8.49	13.42	
Object#mandelbrot						
10.26	37.06	5.23	459076	0.01	0.01	Float#*
9.06	41.68	4.62	465558	0.01	0.01	Float#+
4.06	43.75	2.07	235102	0.01	0.01	Float#**
3.00	45.28	1.53	121301	0.01	0.01	Float#<
2.86	46.74	1.46	114818	0.01	0.01	Float#-

2.73	48.13	1.39	117551	0.01	0.01	Math.sqrt
2.27	49.29	1.16	118569	0.01	0.01	Fixnum#<
2.14	50.38	1.09	114819	0.01	0.01	Fixnum#+
0.78	50.78	0.40	1	400.00	50990.00	
Object#mandel_calc						
0.24	50.90	0.12	3750	0.03	0.05	
Kernel.printf						
0.12	50.96	0.06	3850	0.02	0.02	IO#write
0.06	50.99	0.03	3751	0.01	0.01	Fixnum#-
0.00	50.99	0.00	2	0.00	0.00	
Module#method_added						
0.00	50.99	0.00	100	0.00	0.00	
Kernel.print						
0.00	50.99	0.00	50	0.00	0.00	Float#>
0.00	50.99	0.00	1	0.00	0.00	Fixnum#>
0.00	50.99	0.00	1	0.00	50990.00	#toplevel

图 10-16 图 10-15 中改善后的代码的 profiler 结果，与图 10-13 比起来，方法数减少了

数一数实际的方法调用次数吧。profile 结果第 4 列的合计（减去 top level 的 1）是方法调用的总次数。mandel1.rb 中是 5 248 464 次，mandel2.rb 中是 1 762 049 次，实际上后者是前者的 1/3。由此可以得出 Ruby 高速优化的规则 2。

规则2：减少方法调用

方法调用中，存在多态性这一面向对象的本质特征，先要评价参数，判定对象种类，然后选出一个合适的方法，再将控制移交给该方法，这是一个缓慢的处理过程。

为了避开方法调用，尽可能不用 Ruby 中实现的方法。Ruby 中实现的方法，几乎所有的情况都是通过调用其他方法来实现的。也就是说，只要使用了一次 Ruby 中实现的方法，就会有多余的方法调用发生。

10.2.5 高速优化之二：利用立即值

Ruby 中有几类对象并不实际分配内存，而是用引用（reference）本身来表示，这种值称为立即值（immediate value）。

现在的 Ruby 中，小的整数（ $\pm 2^{30}$ 以内）、真假值、nil 和符号名等都是立即值。

立即值既然不生成对象，就不用担心对象的生成成本，而且也不用垃圾收集处理，所以它具有特别理想的特性。

曼德勃罗集合的计算是浮点小数的计算，为了将其整数化，需要将其“抬高”，变成固定小数点数。固定小数点以后的 `mandelbrot` 方法如图 10-17 所示。

```
def mandelbrot(c_r, c_i)
  limit=95
  iterations=0
  cr = (c_r * 100).to_i
  ci = (c_i * 100).to_i
  zr = zi = 0
  while iterations<limit and Math.sqrt(zr*zr+zi*zi)<1000
    zr, zi = (zr*zr-zi*zi)/100+cr, (zr*zi+zi*zr)/100+ci
    iterations+=1
  end
  return iterations
end
```

图 10-17 `mandelbrot` 方法（改善版 2）

这里为确保小数点以下两位，将值放大 100 倍。在乘法运算后，因为结果被重复放大了，所以除以 100。

`mandel2.rb` 中 `mandelbrot` 方法被替换后，请保存为 `mandel3.rb`。

```
% time ruby mandel3.rb > /dev/null
real    0m0.467s
user    0m0.464s
sys     0m0.004s
```

与 `mandel2.rb` 相比，性能仅改善了一点点。所以，说老实话，改善没有期望的那么大。在 `mandelbrot` 程序中，生成的对象数比想象得少，垃圾收集的负担也没那么大。固定小数点所需的“抬高”运算增加了方法调用，这抵消了利用立即值带来的好处。如果在分辨率更高一点的计算中，垃圾收集的负担会更高一些，或许会得到更大的差异。

固定小数点的计算也有副作用。仔细观察 `mandel13.rb` 的输出，可以看出图形的细微部分略有差异。这是因为计算到小数点以下两位，计算结果有偏差。

10.2.6 高速优化之三：利用C语言

因为 Ruby 是解释性语言，单纯计算的循环不是很快。如果将处理交给编译器，就可以变得很快。根据 `profile` 的测定结果，如果找到了瓶颈所在，就可以将该部分用 C 语言来实现，这也是一个有效的战略。

与其他语言比较起来，用 C 语言来编写 Ruby 扩展库相当简单。

这次没有详细说明扩展库是有原因的，因为有更好的方法。

RubyInline 就是更好的方法。简单地说，RubyInline 就是 Ruby 程序中能够直接嵌入的 C 语言代码。初次执行时，C 语言代码被自动编译，第 2 次以后的执行会自动装载缓存中的扩展库。或许，看看实际的代码更容易懂。

安装 RubyInline 后，请在程序先头加上 `require "inline"`。

于是，将 `mandelbrot` 方法换成图 10-18 中的代码。

```
class Object
  inline do |builder|
    builder.include "<math.h>"
    builder.c "
      int mandelbrot(double cr, double ci){
        long limit = 95;
        long iter = 0;
        double zr = 0, zi = 0, zzr, zzi;
        while (iter < limit && sqrt(zr*zr+zi*zi) < 10){
          zzr = zr*zr-zi*zi+cr;
          zzi = zr*zi+zi*zr+ci;
          zr = zzr; zi = zzi;
          iter++;
        }
        return iter;
      }"
  end
end
```

图 10-18 mandelbrot 方法 (RubyInline 用)

图 10-18 中，用 `builder.include` 声明必要的包含文件，用 `builder.c` 定义 `Object` 的方法。方法本身用 C 语言编写，处理内容与 `mandel2.rb` 相同。

置换后的文件保存为 `mandel4.rb`，然后执行。第 1 次执行需要在内部进行编译，会稍微多花些时间，第 2 次以后，编译结果已经保存下来，这样就可以高速执行。与以前一样，执行 10 次，其中花费时间最短的结果如下。

```
% time ruby mandel4.rb >
/dev/null
real    0m0.083s
user    0m0.076s
sys     0m0.008s
```

0.083 秒！与原始程序相比快了 31 倍，与使用同一算法进行高速化的 `mandel2.rb` 相比也快了 6 倍以上。

与 RubyInline 制作扩展库的方案比起来，这要简单易行得多，是一种更优秀的提速方案。但是，如果没有编译环境，当然就不会运行，所以需要稍稍选择一下执行环境。而且，如果是 Linux，那么很少会出问题。如果是 Windows，则只能在 Cygwin 环境下运行。

10.2.7 高速优化之四：采用合适的数据结构

这是不是到了极限了呢？已经用 C 语言实现了，或许更进一步的高速化就很难了。

但仔细看看这个程序，是不是真的有必要对 `mandelbrot` 方法进行如此频繁（3650 次）的调用呢？如果能够削减调用次数，或许就可以更高速地处理。

为此，好像需要变更数据结构。图 10-19 中，用了表示均质数值的数组 `NArray` 类。

```

require 'narray'

def mandel_calc(min_r, min_i, max_r, max_i, res)
  h = Integer((max_i - min_i) / res).abs
  w = Integer((max_r - min_r) / res).abs

  c = (NArray.scomplex(w,1).indgen!/25+min_r) +
      (NArray.scomplex(1,h).indgen!/25+min_i)*1.im
  z = NArray.scomplex(w,1)+NArray.scomplex(w,1)*1.im
  a = NArray.sint(w,h)
  idx = NArray.int(w,h).indgen!

  limit=95
  i=0

  while i<limit
    z = z**2+c
    idx_t,idx_f = (z.abs>10).where2
    a[idx[idx_t]] = i
    break if idx_f.size==0
    idx = idx[idx_f]
    z = z[idx_f]
    c = c[idx_f]
    i+=1
  end
  h.times do |y|
    print "|"
    w.times do |x|
      printf "%c",126 - a[x,y]
    end
    print "|\\n"
  end
end

mandel_calc(-2 , -1, 1, 1, 0.04)

```

图 10-19 使用 NArray 的曼德勃罗集合的计算

将这个 程序 保存为 mand-el5.rb，并执行一下。

```

% time ruby mandel5.rb > /dev/null
real    0m0.056s
user    0m0.048s
sys     0m0.008s

```

比使用 `RubyInline` 的程序执行时间更短，完全是一眨眼的工夫。改变数据结构和选择计算方法，能取得很大的效果。

但是，`mandel5.rb` 的执行结果，细微部分与曼德勃罗集合有很大的差异。本来想与其他的程序得出同样的结果，但 `NArray` 的使用方法弄得还不是很懂，时间就过去了。

10.2.8 全部以C语言计算

作为参考，试一试全部以 C 语言实现会高速到什么程度吧。算法已经知道了，只是机械地换为 C 语言。结果如下所示。

```
% time a.out > /dev/null
real    0m0.007s
user    0m0.004s
sys     0m0.004s
```

确实很快。像这种单纯的计算，怎么也敌不过 C 语言。但这个 C 程序之所以能够很简单地开发出来（5 分钟左右的工作），是因为之前已经存在能够通畅运行的 Ruby 程序。在开发能够高速运行的程序的时候，首先用 Ruby 写一段能够运行的代码，然后在正式环境中用 C、C++ 或 Java 等重写，这也是一个可行的途径吧。

10.2.9 还存在其他技巧

这次没有尝试的，还有“以空间换时间”的技巧。也就是将计算的中途结果保存起来，避免多次重复同样的处理，结果让处理能够在短时间内完成。在曼德勃罗集合的计算中虽然不适用，但在反复进行同样计算的处理中，却能够发挥极大的作用。

* * *

以上学习了性能优化的具体技巧。将这些技巧再度总结一下吧。

- 根据测定，发现瓶颈。
- 减少对象。

- 减少方法调用。
- 避开用 Ruby 实现的方法。
- 使用立即值。
- 瓶颈部分用 C 语言记述。
- 以空间换时间。

如果你正在因为自己的程序太慢而苦恼，这些技术可能会起些作用。但也不要忘记性能优化的格言——过早的优化是万恶之源。

10.3 并行编程

在 Linux、Windows 等现代操作系统中，多个程序能够同时运行。比如，可以一边浏览网页，一边在后台刻录 DVD-R。

计算机虽然只有一个 CPU，但操作系统能够将程序的执行单位细化，然后分开执行，从而实现伪并行执行¹。最近，有称为多核的，即一台机器上有多个 CPU（CPU 核心）同时运行。

¹ 这种伪并行执行称为并发（concurrent）。使用多个 CPU 真的同时执行称为并行（parallel）。

10.3.1 使用线程的理由

多个程序同时运行的时候，操作系统以进程²为单位同时运行。若各个进程完全独立，也就无二话可说。但是，当多个进程协调起来进行处理的时候，就有点麻烦了。因为进程间必须共享信息。

² 配置了专用内存空间的程序执行代码称为进程，进程中也包含 CPU 状态。

假设某一进程（父进程）启动了别的进程（子进程）。进程之间具有独立的内存空间，子进程中的变量值更改了，也不会影响到父进程³。

3 子进程管理的各种变量值，是从父进程复制而来的。子进程的变量值改变了，并不能反映到父进程中去。

这样，进程之间就有必要进行通信。只能用管道⁴ 或套接字⁵ 以字节流⁶ 的形式传送信息。**UNIX** 和 **Linux** 中还有共享内存⁷ 这一机制，但使用起来并没那么容易。

4 **pipe** (`|`)，是把某一程序（进程）的标准输出传给另一进程，作为其标准输入的一种参数传递方法。用法就像 `cat file | more` 这样。

5 **socket**，是像操作文件一样，处理网络通信的一种机制。

6 所谓字节流，是单向或者双向数据通信时的一个概念。发信端和收信端数据的排列是一致的。

7 共享内存是指多个进程之间能够共享的内存。虽然高速，但缺点是需要有防止冲突的机制，不易取得更新时机。

内存空间之所以独立，是因为要保护进程，避免进程之间互相干扰。但当进程之间通信的数据量很大，或者结构很复杂的时候，就有点麻烦了。

于是就使用线程⁸。使用线程可以在一个进程中同时进行多个处理。**thread**（线程）原意是缝衣线的意思。执行流程像缝在衣服上的线一样，一会儿出现，一会儿消失（一会儿执行，一会儿停止）。或许名字也起源于这种联想吧。

8 一个进程由一个以上的线程构成。

与进程不同，同一进程的线程可以共享内存空间。所以，不同的线程能够引用同一对象。不需要经过将数据变为字节流再进行通信这样麻烦的处理，就能交换信息。

Ruby 中线程机制以标准形式提供。**Ruby 1.8** 中实现的线程称为用户级线程。这与操作系统提供的内核级线程不同，不能够活用多个 **CPU**。而且，线程之间的切换成本很高，会耗费一部分处理性能，对机器的

处理性能不利。但是，不管操作系统有没有提供线程功能，任何操作系统（包括 MS-DOS）上动作都相同。

Ruby 1.9 中，使用 `pthread`——POSIX 标准中规定的操作系统提供的线程功能。

能够轻易进行并行编程，是 Ruby 的优点之一。

10.3.2 生成线程

现在来看看用 Ruby 进行线程编程。图 10-20 是一个使用线程的程序示例⁹。

⁹ 本例中，线程之间不共有信息。

```
Thread.new{
  loop{
    puts "thread 1"
    sleep 2
  }
}

loop {
  puts "main thread"
  sleep 3
}
```

图 10-20 单纯线程程序的例子

`Thread.new` 生成一个新的线程，该线程执行其下面的一段程序（第 1 行至第 6 行）。请注意，这段程序内容和 `Thread.new` 之后的内容（第 7 行以后）同时执行。最初的线程每隔 2 秒输出一串 `thread 1`。

随着线程的生成，程序的执行分成了两个流程。一个不停地显示 `thread 1`，另一个本来的控制流程（被称为主线程），继续不停循环其 `loop`，每 3 秒显示一串 `main thread`。

让程序运行一会儿，厌烦了就按 `Ctrl+C` 中断。执行如图 10-21 所示。

```
thread 1
main thread
thread 1
main thread
thread 1
thread 1
main thread
thread 1
main thread
thread 1
-:14:in `sleep': Interrupt
      from -:14
```

图 10-21 图 10-20 中程序的执行结果“-:14:in ...”以下是执行中断时的显示内容

在 Thread 类中，有表 10-3 所示的 Thread.new 等类方法（class method），还有后面将会用到的如表 10-4 所示的实例方法（instance method）。

表10-3 Thread 类的类方法

方 法 名	功 能
Thread.abort_on_exception	异常时中断进程
Thread.abort_on_exception=val	中断状态的设定
Thread.current	当前线程
Thread.exit	当前线程终止
Thread.fork	new的别名
Thread.kill	当前线程终止
Thread.list	线程一览
Thread.main	主线程
Thread.new	线程的生成
Thread.pass	执行权的转让
Thread.start	new的别名
Thread.stop	一时停止

表10-4 Thread 类的实例方法

方 法 名	功 能
-------	-----

th[key]	线程固有数据
th[key]=val	固有数据的设定
th.abort_on_exception	异常时进程中断
th.abort_on_exception=val	中断状态的设定
th.alive?	确认是否处于活动状态
th.exit	线程终止
th.join	等待线程终止
th.key?(key)	确认key对应数据的有无
th.keys	key的一览
th.kill	线程终止
th.priority	优先度
th.priority=val	优先度的设定
th.raise(exception)	异常的发生
th.run	使线程执行
th.safe_level	安全级别
th.status	显示线程状态
th.stop?	确认是否停止
th.value	线程的返回值
th.wakeup	唤起线程

10.3.3 线程的执行状态

Ruby 的线程有以下 4 种状态。所有线程最初都从执行（run）状态开始。各种状态按图 10-22 进行状态迁移。

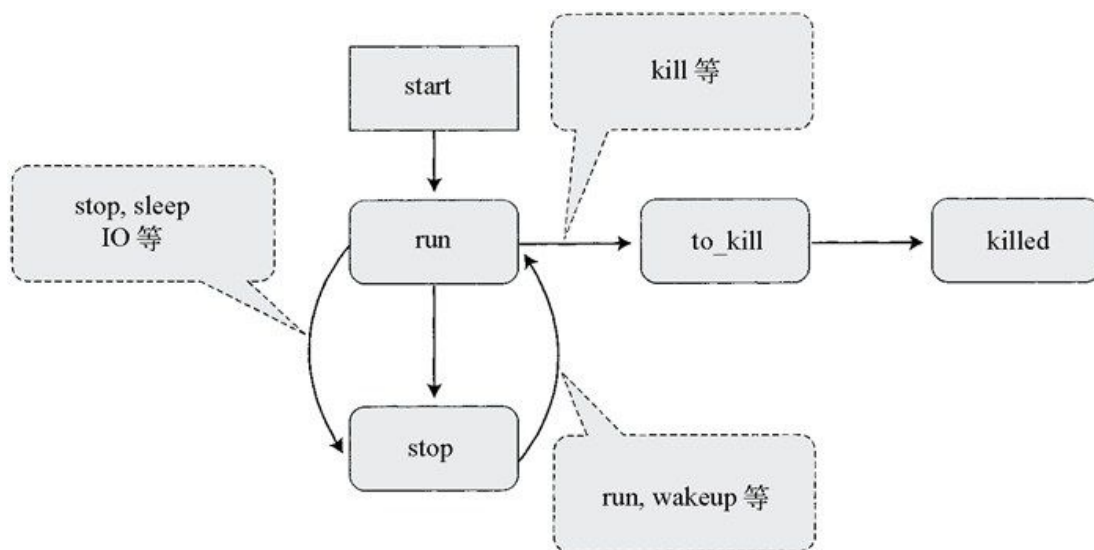


图 10-22 线程处于 4 个状态中的一种

run: 执行中

严格来说，Ruby 的线程将处理细分成一段一段，交替执行，与其说执行中，不如说执行可能更贴切。

stop: 停止中

停止的理由有 IO 等待、时间等待和 join 等待。IO 等待是指等待外部输入的状态。时间等待是指到达指定时间为止，停止的状态（sleep 等的结果）。join 等待是指表 10-4 所示的 join 方法里，等待别的线程终止的状态。

to_kill: 终止处理中

线程被强制终止，完全终止之前，正在处理 ensure¹⁰ 等的状态。比如文件关闭处理等。

¹⁰ 所谓 ensure，是指作为后处理中执行的程序块。

killed: 终止

终止处理完成之后的状态。一旦终止之后，线程不能再复活。终止之后的线程，也不会列在 Thread.list 方法所显示的一栏中。

调查线程现在的状态，有 3 个方法，分别是 **status**、**alive?** 和 **stop?**。**status** 返回线程现在状态的字符串。若线程处于活动状态，则返回 **run**、**sleep** 或 **aborting** 中的某一个。若线程处于 **killed** 状态，则返回 **false** 或者 **nil**。

alive? 在线程处于活动状态的时候返回 **true**。**stop?** 在线程终止的时候返回 **true**。终止包含 **killed** 状态。

对于因异常而终止的线程，调用表 10-4 中的 **join** 方法或 **value** 方法，能够再产生一次导致此线程终止的异常。对于检查线程中是否发生了异常，以及线程对应的异常情况，这很有用。

```
t = Thread.new{...}
t.join      # 等待t 的终止
            # 若有异常，再发生
```

join 方法等待线程的终止。**value** 方法在等待线程终止的同时，还要返回（该线程程序块的）最后的计算值。

10.3.4 传递值给线程的方法

现在再来看 一个使用线程的程序吧（参见图 10-23）。

```
require "socket"

gs = TCPserver.open(0)
loop do
  # 线程的启动
  # start 参数传递给块参数
  Thread.start(gs.accept) do |s|
    .....处理内容.....
    s.close
  end
end
end
```

图 10-23 使用线程记述的 socket server

图 10-23 中的程序是使用线程的 `socket server` 的雏形。接受了来自客户端的连接以后，就生成一个新的线程，然后将处理交给它。线程是并行运行的，一个客户端的处理完成之前，可以没有任何问题地处理来自下一个客户端的连接。请注意这里给线程传递值的方法。

图 10-23 中，有必要说明一下 `Thread.start` 的参数。传递给 `Thread.start` 的参数被赋值给块参数¹¹，这可用于给线程传递值。

11 块是 Ruby 特有的语法。它是可以追加在方法调用末尾的代码块。其表现形式是方法名{代码}。附加有代码块的方法可以在运行时调用代码块的内容。{代码}中以|变量|形式所声明的变量（如图 10-23 中的|s|）称为块参数。

因为可以从代码块看见外侧的变量，所以变量可以直接引用。比如，程序的循环部分像下面这样替换以后也能运行。

```
s = gs.accept
Thread.start() do
  .....处理内容.....
  s.close
end
```

但时机不好时，会产生误动作。也就是说，在线程处理完成之前，又去执行下一个 `accept` 时，`s` 的值可能被替换。

在线程编程时，各个线程同时运行，动作的预想比通常程序要困难。可以说需要更好的想象力。

10.3.5 信息共享所产生的问题

使用线程，在并行处理中可以简单地实现信息共享。但也并不是光有好事。现实社会也是这样，在共同生活中，如果彼此之间太随便，想干什么就干什么，那么关系迟早要破裂。线程处理也是如此，有可能发生以下问题。

- 数据完整性的丧失。
- 死锁。

不管哪一个，与其说是线程的问题，不如说是并行处理本身的问题。但是，线程要共享内存空间，比其他的并行处理更容易发生问题。下面逐个来看这些问题吧。

10.3.6 数据完整性的丧失

从刚才图 10-23 的例子已经知道，共享中的数据如果被更改了，会发生意想不到的恶劣影响。线程之间共享同一内存空间，可能无意间同时访问了同一变量或同一对象。这就会发生数据完整性的丧失。最常见的例子是银行账户（参见图 10-24）。

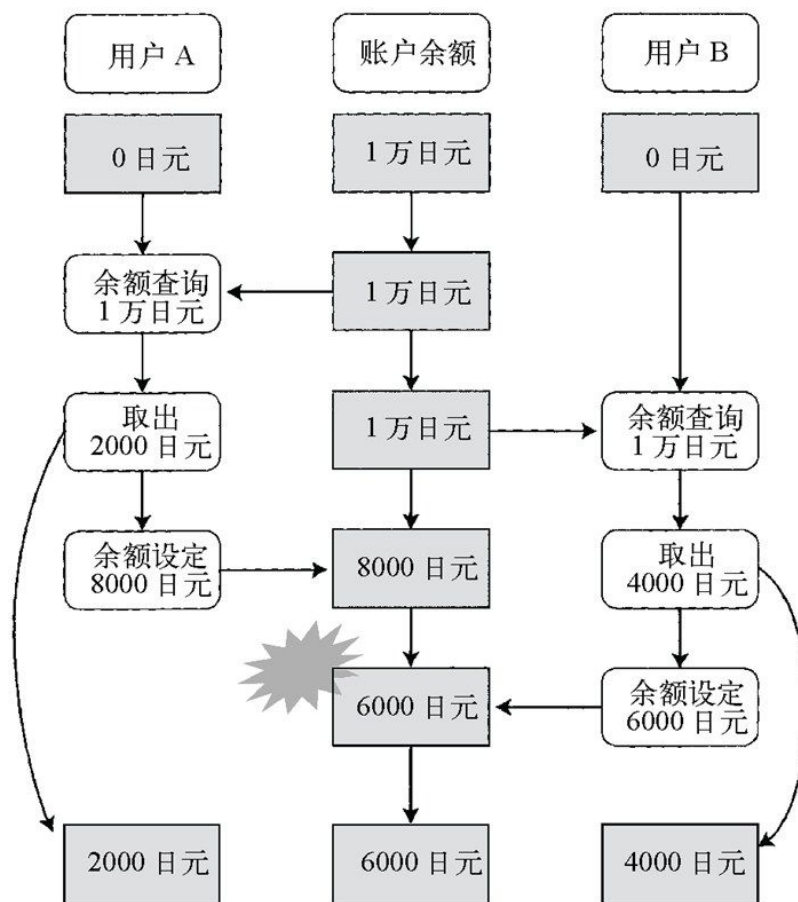


图 10-24 数据完整性的丧失，银行交易的例子

对于某一银行账户，如果用户 A 和用户 B 几乎同时访问时，就会出问题。

简单说明一下流程。用户 A 查询账户余额，取出 2000 日元后，将差额设为新的余额。另一方面，用户 B 也同样查询账户余额，取出 4000

日元，将差额设为新的余额。账户最初只有 1 万日元，图中最后的合计金额（用户 A、用户 B 和银行账户）却变成 1.2 万日元。银行如果真有了这种麻烦，那可是个大问题。

图 10-24 中流程的笨拙之处在于多个线程毫无讲究地访问同一个账户。在至今已经习惯按次序执行的世界里，没有机会同时引用同一数据，不知不觉就会忘了这个矛盾情形。

在并行处理环境里，从余额查询开始到新的余额设定为止，（别的处理）不能中途插入。这样不能分割的处理称为原子（atomic）处理。在原子处理的过程中，需要将此账户保护起来，不能让别的线程对此账户进行操作。

10.3.7 死锁

编程中所说的死锁，是指多个线程互相争夺资源、谁也动不了的状态。经典的案例有哲学家就餐问题。哲学家就餐问题如下所述。

5 个哲学家围坐在一个圆桌旁（参见图 10-25）。每个哲学家前面放一个盛满意大利面条的盘子。每个盘子两边各放一根筷子（共 5 根）。哲学家从早到晚都在思索，有时肚子饿了，就从盘子两边拿筷子吃面条。哲学家举止得体，即使肚子再饿，也不会用手或单根筷子吃面条。

突然，在某一瞬间，所有哲学家都要吃面条，假设都拿了自己右边的筷子。再想拿自己左边筷子的时候，发现已经被别的哲学家拿走了。每个哲学家都是右手拿着筷子，一直等着旁边的哲学家吃完。但是，谁也吃不成，大家都在那儿饿着。

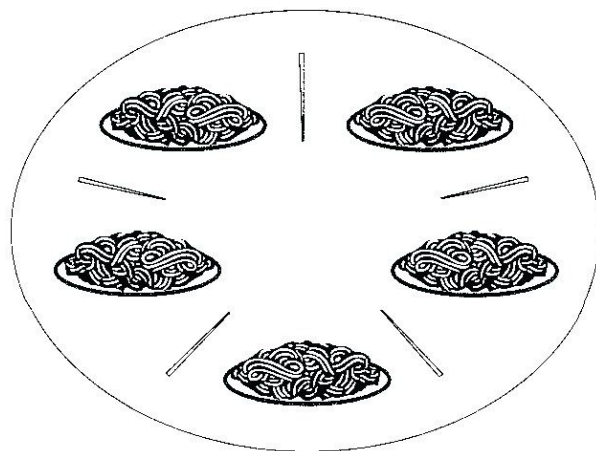


图 10-25 哲学家的晚餐。5 盘意大利面条及 5 根筷子交错排在桌子上

像这样，由于某种竞争，处理永远陷入停止状态，我们称之为死锁。

多个并行处理共享的资源（称为 **resource**，这种情况下是筷子）不足的时候，如果访问步骤考虑不周的话，就会引起死锁。

由上述哲学家的情况，我们可以知道问题出在资源的访问顺序上。不仅限于此问题，在很多场合，访问顺序的整理都是很重要的。比如说，所有哲学家，如果能贯彻执行以下规则，就可以规避死锁¹²。

12 严格地说，全体成员的动作有可能同时发生，所以还必须考虑放下筷子的时刻。

- (1) 首先取右边的筷子。
- (2) 如果左边的筷子拿不到的话，就放下右边的筷子。

10.3.8 用锁来实现对资源的独占

银行的例子也好，哲学家的例子也好，如果同时访问资源，就会成为程序的陷阱。

规避资源的同时访问，有几种方法，其中最简单的是独占使用中的资源。为此，提供了资源的“锁”这一手段。

现实社会里，最常见的“锁”，大概就是厕所的单间吧。厕所的单间，一次只能供一个人使用。有人进去了，就加上锁。从外面看是红记

号，就知道有人在里面。线程处理也是一样，使用仅有的资源的时候，就加上锁，加上一个“独占中”的记号就行了。

银行账户的例子中，对账户进行一连串操作的过程中将账户加上锁，中途不允许别的操作挤进来，就可以解决死锁问题。哲学家的例子中，在抓住两边筷子时加上锁，使得别的哲学家不能干扰，就能够规避死锁问题。

Ruby 里，加锁用 **Mutex** 类。**Mutex** 是互斥锁（Mutual Exclusive Lock）的缩写。使用 **Mutex** 类，需要 **require thread** 库（参见图 10-26）。

```
require 'thread'

m = Mutex.new

...
m.synchronize {
  ...# 使用资源的处理
}
```

图 10-26 Mutex 类的使用示例

Mutex 类的 **synchronize** 方法，用于处理该代码块时，对 **Mutex** 对象进行加锁¹³。别的线程想要对同一个 **Mutex** 对象执行 **synchronize** 方法时，必须要等到现在执行中的 **synchronize** 方法完成并解锁之后才能进行。**synchronize** 围起来的代码块，是别的线程不能侵入的领域，有时称为 **critical section**。

13 代码块执行后，**Mutex** 的锁自动解放。

不能忘记的是，**Mutex** 无非是一个君子协定。实际上不检查锁就可以操作资源，并非自动禁止。用锁来保护资源的时候，别忘了对所有资源的访问都要检查锁。

为了使用 **Ruby** 让这个检查变得聪明些，对资源的访问全部要经由某一特定的对象，由这个方法在内部对锁进行检查（参见图 10-27）。

```
class Resource
  def initialize
    @mutex = Mutex.new
  end
  def use
    @mutex.synchronize{
      ...
    }
  end
end

r = Resource.new
r.use
```

图 10-27 访问资源的专用类之记述示例

Java 中，方法定义声明为 **synchronize**，该方法被调用时自动加锁。

10.3.9 二级互斥

很遗憾，锁的问题并不全是 **Mutex** 所能覆盖的单纯事例。比如数据库中，对数据的访问需要以下多种互斥控制。

- 可以同时引用。
- 禁止同时更新。
- 禁止更新中引用。
- 禁止引用中更新。

引用与更新这种两个层次的互斥（二级互斥），在文件读取时也经常会发生。

现实世界中也有类似例子。比如看电视和更换频道相当于二级互斥，如果有个人在看电视，就不让别人看同一个频道，这种独占就太过分了。但如果没有任何互斥控制，正看在劲头上，别人却换了频道，就让人讨厌了。

这种情况下，使用 `sync.rb` 中定义的 `Sync` 类。`Sync` 类和 `Mutex` 同样使用，但有一个区别。`synchronize` 方法可以指定互斥模式。

互斥模式有两种，一种是 `EX` 模式，一种是 `SH` 模式。`EX` 模式像更新一样，同时只能有一个执行。`SH` 模式与 `EX` 模式虽然不能同时执行，但与其他模式并不互斥。这两种模式分别以 `:EX` 和 `:SH` 指定。

现在，把 `Sync` 类看做电视机吧（参见图 10-28）。编程内容有点不自然，将就一下吧。正如 `Mutex` 所示，资源电视机）对象的方法 `start_watch`，`end_watch`，`set_ch` 中嵌入排他控制。这样更明智一些。

```
require 'sync'

class TV
  def initialize
    @ch = 1
    @sync = Sync.new
  end
  # 开始看电视
  def start_watch
    @sync.lock(:SH)
  end
  # 看完电视
  def end_watch
    @sync.unlock(:SH)
  end
  def set_ch(ch)
    @sync.synchronize(:EX){
      @ch = ch # 换频道
    }
  end
end
```

图 10-28 使用 `Sync` 类的互斥模式解决看电视问题的示例

图 10-28 中的 `TV` 类里，有人在看电视的时候，不能更换频道。只有在没人看电视的时候，才能给予频道变更权。这样做，就避免了有人独占电视，或者有人正在看电视时更换频道的事态。

10.3.10 用队列协调线程

使用锁对资源进行互斥控制，是线程间保证协调的一种机制。除了锁以外，为了保证协调，还有别的方法。

本来问题的根源就在于多个进程访问同一资源，如果不进行这种访问就好了。这种方法，因为没有共享的东西，而被称为无共享（**shared nothing**）。

为了实现无共享，给每个资源准备一个管理用的线程，各线程间只能交换某些限定的信息。

线程间信息交换的方法有很多种，有代表性的有消息存储（**message banking**）、信道（**channel**）及队列（**queue**）。

这里介绍通过队列进行的线程间通信。在无共享结构里，某一线程制作数据，另一线程通过队列获取数据，然后进行下一步处理。

这种关系称为生产者—消费者模型，制作数据的线程称为生产者（**producer**），获取数据的线程称为消费者（**consumer**）（参见图 10-29）。有的消费者又将数据进行加工，传递给别的消费者，承担着生产者的角色。

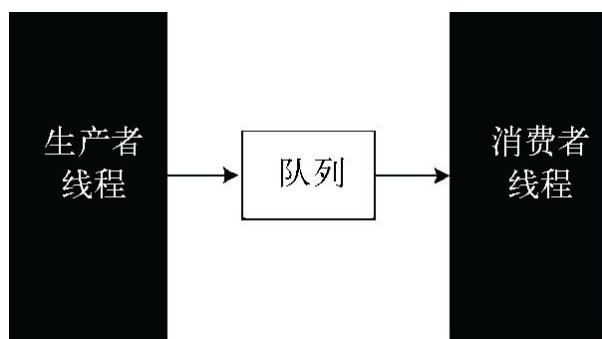


图 10-29 生产者—消费者模型，通过队列进行线程间通信

连接生产者和消费者，起着“传送带”作用的是 **Queue** 类。队列是具有 **FIFO**（**first-in, first-out**，先进先出）性质的数据集合，恰如其名，先进进来的数据先出去。

来看一个实际使用 **Queue** 类的程序吧（参见图 10-30）。

```
require 'thread'
```

```
q = Queue.new

producer = Thread.new {
  10.times {|i|
    q.push(i)
    sleep(1)
  }
  q.push(nil)
}

consumer = Thread.new {
  loop{
    i = q.pop
    break if i == nil
    puts i
  }
}
consumer.join
```

图 10-30 Queue 类的使用示例

生产者每秒往队列里追加（**push**）一个整数，消费者从队列里取出（**pop**）数字输出。消费者速度快，队列变空之后，消费者暂停¹⁴，直至生产者往队列里追加数据。

14 图 10-30 中的场合，因为有 **sleep(1)**，所以生产速度 < 消费速度。

生产者送出一定个数的整数之后，会送出一个表示终止的标识 **nil**。消费者接收到 **nil** 后，用 **break** 跳出循环。主线程通过程序最后一行的 **join**，等待消费者线程的执行。执行结果如图 10-31 所示。

```
% ruby q.rb
0
1
2
.....
8
9
```

图 10-31 图 10-30 中程序的执行结果

那么，假设生产者比消费者快，将会怎样呢？来不及消费的数据积压在队列里，队列的长度会越来越长。如果来不及消费，也许就没必要全力生产数据。

这种情况下，便利的是 **SizedQueue** 类。生成一个长度有一定限制的队列，数据积累到一定数量，就停止追加数据¹⁵。**SizedQueue** 的使用方法很简单，在刚才的程序里，只要把下面的第 1 行换成下面的第 2 行即可。

15 这样就不会丢失来不及消费的数据。

```
q = Queue.new
q = SizedQueue.new(size)
```

size 部分放入队列的最大长度。不管是生产太快，还是消费太快，都能自动调整。

队列也可用于解决资源的竞争。准备一个用于管理资源的线程，对资源的请求（request）通过队列发送给该线程（参见图 10-32）。

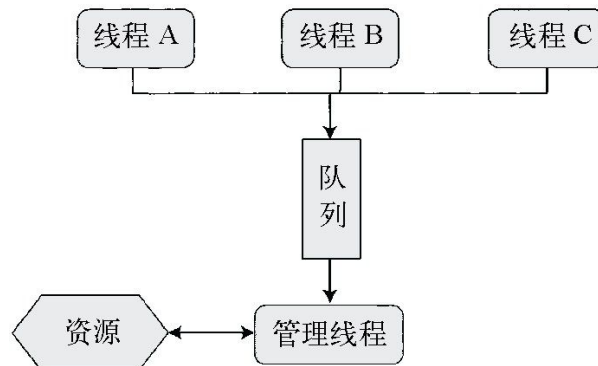


图 10-32 用队列解决资源竞争的方法

这样，直接操作资源的只有管理线程，因此不会发生竞争。

有一个实例。**Ruby/Tk**¹⁶ 采用这种方法。多个线程不能同时访问通过 Tk 的 GUI 操作，来自各个线程的 GUI 命令都通过队列发送到管理线程。管理线程调用 Tk 库来处理发送过来的 GUI 命令。

16 所谓 Ruby/Tk，是 Ruby 与 GUI toolkit Tk 组合而成的处理系统。

10.3.11 锁模型与队列模型的比较

如果与线程相关的程序出了问题，那么排错（debug）起来可就难了。与线程相关的程序错误（bug）往往因为时机不同，有时发生，有时不发生。即使执行同样的程序，错误要么不能再现，要么 10 次之中只有 1 次不行，成为性质恶劣的程序错误。这是软件错误中最难排除的一类。

这次特意没有同时访问多个资源，回避了线程编程中容易出现的问题。但是，资源的访问说起来简单，事实上，因为含有方法调用、变量引用等许多琐碎的处理，难免考虑不周。

作为防止资源竞争的手段，按顺序分别介绍了使用锁的锁模型和使用队列的队列模型。不能说哪种更优秀，它们各自有以下特征。

锁模型

如果竞争足够少，多数情况下能保持较高性能。但是，对于资源的竞争，不能忘记加锁。要做到完美无缺较难。

队列模型

在竞争少的时候，其性能比不上锁模型，但对于程序员来说，它比锁模型更容易贯彻。虽然各种处理系统的表现不尽相同，但这种方法很少会因为线程增多而带来性能低下的恶果。

最近，有一种语言 Erlang，在一部分人中成为话题。该语言采用了与队列模型本质上等价的消息存储。Erlang 中，即使线程（Erlang 中称 process）的数目很多，也很少会有性能低下的现象，多核时还能最大限度地发挥多个 CPU 的优势。今后，多核或具有扩展性的队列模型或许会成为主流。

随着 CPU 电路的微小化，性能的提高也会达到其物理极限。在不远的将来，CPU 的热密度或将达到火箭喷射口的程度¹⁷，而且，根据光速极限，1 个时钟周期内电子能够移动的距离也只能达到几厘米的水平。

17 现在的 CPU 的热密度也比热板高。热板是吃烤肉时用于加热的铁板。

多核能够超越此极限，想要达到更高的性能，多核成为必然的途径。在多核环境下，如果软件还像现在这样，就不能充分享受 CPU 的性能提高所带来的好处。

为了达到高性能，软件必须提供对并行编程的支持。所以，对编程语言来说，并行处理今后肯定会成为非常热门的话题。关于性能和扩展性，现在还是 Ruby 的弱点，是今后要解决的课题。

10.4 前景可期的并行编程技术，Actor

前一节我们已经看到，线程编程容易变得复杂化，不但因为同步和竞争容易引起问题，而且一旦发生问题，症状会因时机不同而变化，以致于很难处理。这或许是因为线程这种并行化处理的模型不适合人类去处理吧。

并行编程，要求有更高的抽象度和对人类而言更简单的编程模型。Curl Hewitt 提倡的“Actor Model”（参与者模式）或许会成为这个问题的答案。

10.4.1 何谓Actor

前言就说这么多。还是来看 Actor 是什么吧。所谓 Actor，是（仅）通过消息（message）进行通信的实体。

单从这个定义来看，好像与面向对象语言的对象也没什么区别，但其实还是有的。向对象发送消息（方法调用），调用开始后，会一直等到返回结果，是一种同步方式。而向 Actor 发送消息，仅仅是发送消息而不等返回结果，是一种异步方式。

作为并行处理的机制，大家都知道线程。多个控制流同时运行的线程，非常简单和容易实现，如果协调和同步方面不需要花费很大成本的话，就能够得到很高的性能。

另一方面，Actor 基本上仅仅通过消息进行信息交换。因为不能直接共享同一个值，信息传达上就要多花些代价。

但是，为了避免线程间资源竞争¹，需要动用锁或者队列等各种方法来保护对资源的访问。

¹ 所谓资源竞争，是指多个线程同时访问同一资源（变量、设备等），由于时机不同，执行结果变得不确定，或者程序状态变得不完整。

Actor 由于没有消息以外的信息传递手段，所以不用担心 **Actor** 之间的资源竞争。发送给 **Actor** 的消息，都配送到各个 **Actor** 所拥有的邮箱里。多个消息同时到达时的竞争，已经由内嵌到系统中的排除机制来处理了。

Actor 有一个大的优点是安全，但更大的优点是“易懂”。**Actor** 根据消息进行处理，必要的话，会向其他 **Actor** 传递消息，或者向源 **Actor** 返回消息。

这与现实世界中人与人之间的相处没有大的差别。现实世界中，别人在想什么你不知道，想要求什么时，需要通过某种手段传达“消息”。

另外，通过消息接受某种要求的人，与提出要求的人独立开展工作。工作完成之后，发送“完成”的消息返回给提出要求的人。对于我们而言，这种方法非常自然，整体处理流程也容易把握。**Actor** 原意是演员，正如这个字面意思，**Actor**（演员）与其他 **Actor**（演员）通过台词（消息）对白，完成他的角色，将剧情（程序）进行下去。

简单总结一下，理论上要达到最高性能，一般认为线程更优秀。但如果不注意使用线程的话，会出现与时机相关的很麻烦的问题。在计算机性能日渐提高的今天，与理论上最高性能的可能性相比，由 **Actor** 实现的安全性和易懂性更引人注目。

10.4.2 操作**Actor**的 3 种处理系统

下面介绍操作 **Actor** 的 3 种处理系统。为了将处理系统的区别突出出来，我们使用共同的例题。例题是名叫 **pingpong** 的程序。该程序进行以下处理：

1. 生成两个 **Actor**；

2. 首先，向一个 Actor 发送循环次数和包含另一个 Actor 信息的信息；
3. 接收消息的 Actor 将循环次数减 1，返回消息给另一 Actor；
4. 重复以上消息的发送和接收，直至循环次数变为 0。

两个 Actor 互相发送消息，像打乒乓球（pingpong）一样。虽然是特意编造的例题，却能表现各个处理系统如何进行 Actor 最根本的处理，即 Actor 如何生成，消息如何发送和接收。

首先要介绍一下采用 Actor Model 的函数型语言 Erlang。因 Actor Model 的并行编程而最近才受到瞩目的 Erlang 是很古老的语言，于 1987 年左右由瑞典的电话公司爱立信（Ericsson）所开发。

Erlang 这个名字，来源于丹麦的数学家、统计学家及技术学家 Anger Krarup Erlang。读作“阿郎”，北欧人发“阿尔”音发得清晰，把这个字读作“艾尔郎”估计也不会错吧。

Erlang 语言因人而得名，但隐含着“爱立信的语言”这种意思。Erlang 实际上用在像爱立信的交换机程序那样的大规模而且复杂的应用上。Erlang 虽然积累了长年的实践经验，但自从 1998 年处理系统开放源代码以来，Erlang 给爱立信以外的人的印象，仍然是未知的技术。

Erlang 是只允许单一赋值的函数型语言。所谓单一赋值，是指一旦赋值，变量的值就不允许再改变。“变量的值现在变成什么？”这是程序排错（debug）中的重要课题。但在 Erlang 中，能够保证所赋的值一直不变，这一点可以帮助程序员比较容易地找出问题所在。

但是，Erlang 的最大特征还是 Actor Model 编程。Erlang 中，Actor Model 是基本，几乎所有处理都用 Actor。Erlang 中，将 Actor 称为 process。这容易与操作系统的进程（process）混淆，但据说因为是不像线程那样共享状态，所以被称为 process。

10.4.3 Erlang 的程序

那么，看看用 Erlang 写的 pingpong 程序吧（参见图 10-33）。第 1 行的“-module(ping)”是模块声明。Erlang 的程序，总是从模块声明开

始。模块名必须与文件名相同。从这一行可以知道，程序以文件名 `ping.erl` 保存。

```
1 -module(ping).
2 -export([main/1,start/1,pingpong/1]).
3
4 pingpong(Name) ->
5     receive
6         {_,0} ->
7             io:format("~s:done~n", [Name]);
8         {Partner, Count} ->
9             if ((Count rem 500) < 2) ->
10                 io:format("~p: pingpong ~p~n",[Name, Count]);
11                 true -> true
12             end,
13             Partner ! {self(), Count -1},
14             pingpong(Name)
15     end.
16
17 start(N) ->
18     Pong_PID = spawn(ping, pingpong, [pong]),
19     Ping_PID = spawn(ping, pingpong, [ping]),
20     Pong_PID ! {Ping_PID, N}.
21
22 main([A]) ->
23     start(list_to _integer(atom_to_list(A))),
24     init:stop().
```

图 10-33 用 Erlang 写的 pingpong 程序

第 2 行声明是能够从模块之外访问的函数名。名字后面的 `/1`，是参数个数的意思。Erlang 中，允许存在函数名相同而参数个数不同的函数，`/1` 是为了明示这个区别。这与 Prolog 语句的记法相同，说明函数型语言 Erlang 实际上受到了 Prolog 相当的影响。

第 4 行定义 `pingpong` 函数。内容暂且放在后面说明，先注意一下变量都是以大写字母开始的这一点。还有，以小写字母开始的标识符都是符号名（symbol），这也很像 Prolog。

10.4.4 Pingpong 处理的开始

颠倒一下说明的顺序。第 17 行开始的 `start` 函数，是实际上 `pingpong` 开始处理的函数。第 18 行与第 19 行的 `spawn` 生成新的 `process`。`spawn` 只以函数为参数，这里使用的 3 个参数分别是模块名、函数名和传递给函数的参数。

以 `spawn` 生成的新 `process` 为基础，`pingpong` 函数开始执行，`pingpong` 函数以 `receive` 语句等待消息发送来。`spawn` 返回新生成的 `process` 的 ID。

这样就生成了两个 `process`。以向其中一个发送消息开始 `pingpong`。`Erlang` 中用 `!` 运算符向 `process` 发送消息。左边是发送消息的目标 `process`，右边是具体的消息。`Erlang` 中，消息可以使用任意值，但这次发送的消息是 `pingpong` 对象的 `process ID` 和所剩循环次数的 2 倍。

消息发送过来之后，用 `receive` 语句等待着的 `process` 开始运行。`receive` 语句对发送过来的消息进行模式匹配，按消息内容分开处理。

模式匹配中，如果指定的是具体的值（符号名、数值等），就检查与指定值是否一致；如果指定的是变量（以大写字母开始的字符串），就赋予对应的值。模式匹配从上开始，选择最初的匹配进行相应的处理。

当计数器到达 0 的时候，匹配 `{_, 0}`。`_` 能匹配任何字符，在不关心匹配的值时使用。这里重要的只是循环次数为 0，而并不关心对方是哪个过程，所以就用 `_` 来匹配过程名。

当计数器到达 0 的时候，`pingpong` 函数送出 `done` 消息并结束。`Erlang` 中 `io:format` 是带格式输出函数。`C` 与 `Ruby` 中的 `%`，在 `Erlang` 中成为 `~`。

`{Partner, Count}` 对消息进行模式匹配。对象 `process` 的 ID 赋值给 `Partner`，剩下的循环次数赋值给 `Count`。如果除以 500 的余数是 0 或者 1，就输出现在的循环次数，并向 `Partner` 返回消息。`Erlang` 中，`%` 是注释记号，求余数用 `rem` 运算符。除以 500 的余数是 0 或 1 时输出循环次数，是因为想在两方 `process` 中都进行输出。

`if` 语句中的 `true -> true`，是不让编译器输出编译错误的“护身符”。Erlang 中包括 `if` 语句，各种东西都是表达式，而且都是有值的。如果没有相当于别的语言的 `else` 部分的 `true->` 部分，就会被认作是值不确定，从而产生编译错误。实际上这里 `if` 语句的值没有使用，我觉得不这样指定好像也没关系。

第 14 行递归调用 `pingpong` 函数，这称为末尾递归，用于实现循环。实际上函数型语言 Erlang 没有实现循环的语法。想实现循环的时候就用末尾递归。

总觉得无限递归，最终会用光堆栈，从而导致错误。但 Erlang 中会对末尾递归进行特别处理，不用担心。

10.4.5 启动pingpong 程序

现在启动 `pingpong` 程序试试吧。如果安装了 Erlang 处理系统，肯定可以使用 `erl` 命令。`erl` 命令不带参数时，将启动对话环境（参见图 10-34）。

```
% erl
Erlang (BEAM) emulator version 5.6.3 [source] [smp:2] [async-
threads:0]
[kernel-poll:false]

Eshell v5.6.3 (abort with ^G)
1> c(ping).                ←ping 模块的编译
{ok,ping}
2> ping:start(1000).       ←ping 的启动
apong: pingpong 1000
ping: pingpong 501
pong:pingpong 500
ping:pingpong 1
pong:done
3>init:stop().             ←环境的终止
ok
```

图 10-34 erl 的对话环境

假设 `pingpong` 程序保存在 `ping.erl` 文件里。首先用 `c` 命令编译。为了执行 Erlang，必须将源代码编译成字节码解释器 BEAM 能够识别的

格式。编译以后就可以利用 `ping.erl` 中声明的函数。每次修改 `ping.erl` 后，都要用 `C` 命令再次装载。

执行 `start` 函数以后，按参数指定的次数执行 `pingpong`。
`pingpong` 完成指定次数以后，输出 `done`。

当然，不经过 `erl` 的对话环境，也可以直接启动程序（参见图 10-35）。这种情况下，启动 `erlc` 编译器，将源代码直接编译成字节码，然后给命令加选项 `-noshell` 启动非对话环境的 `erl`。

```
% erlc ping.erl
% erl -noshell -s ping main 1000
pong: pingpong 1000
ping: pingpong 501
pong: pingpong 500
ping: pingpong 1
pong: done
%
```

图 10-35 `pingpong` 程序的直接运行

这种情形下，从 `ping` 模块的 `main` 函数开始执行。第 22 行的 `main` 函数从命令行的参数接受循环次数，并担当程序执行完成后解释器的终止处理。`atom_to_list` 函数将符号名变为字符串，`list_to_integer` 将字符串变为数值。另外，`init:stop` 函数结束 `process`。

这次，为了这道例题，我很久以来再次挑战 `Erlang` 编程。由于没有循环，不习惯以发送消息为基本的编程等等，觉得有些难受。还有，老是记不住 `Erlang` 的逗号、分号或句号该如何分开使用，错误频发。但一旦习惯了这独特的编程风格，我觉得生产效率也会很高。

10.4.6 `Erlang` 的错误处理

如果有大量的 `process`，就总会有因为程序错误、异常数据输入，还有其他很多预测不到的理由，导致 `process` 异常终止。正在进行大量数据处理的时候，总是想避免因为一个异常而将全部处理作废的事情。对

于有扩展性的系统，重要的是，即使发生了异常事态，也不至于全体系统都停止。

Erlang 中，通过发送消息来通知异常终止。通过 link 机制将 process 与 process 链接（link）起来。被链接起来的 process 在终止之前，往链接目标发送消息说“我要死了”。收到消息的 process 料理死去的 process 的后事。

有了这种机制，使得 Erlang 适合于构筑抗障碍性强的系统。这不禁让人想起 Erlang 长年使用于电话交换机这样重要领域的骄人业绩。

10.4.7 Erlang的使用场所

Erlang 在通常的文本处理中并不怎么快，比如说，用 Ruby 进行的处理，全部移植到 Erlang 上去，基本上没什么好处。Erlang 的好处还是其扩展性。对于多个处理并列执行的情况，分割成合适的 Erlang process，能够发挥多 CPU 的威力。

同样的多任务分割虽然用操作系统的进程或线程也能实现，但 Erlang 的 process 与操作系统的进程或线程相比，能够轻量实现（1 个 process 最小只耗费 300 字节），即使有大量 process 也不必顾虑，尽管生成就是了。更进一步说，Erlang 设计思想不用 process 连基本处理都不能实现，process 分割在某种意义上可以说是强制性的，这样更加有利于发挥语言的特长。

像服务器端的处理一样，Erlang 特别适合于对大量的请求进行非同步处理的任务。Erlang 在 20 世纪 80 年代就已经诞生，但直到最近才变得受人瞩目，这可能是最近发现了它适合于现代服务器端程序这一特点了吧。

10.4.8 面向Ruby的库“Revactor”

进行 Actor Model 编程，如果采用 Erlang 这样独特的语言，就有点太为难了。因为学习新的语言，再移植，是要花费成本的。对于习惯了 Ruby 的人来说，如果能够用平时习惯的 Ruby 来进行 Actor Model 编程，实在是谢天谢地。

能够满足这种需求的是面向 Ruby 的 Actor 库 Revactor。Revactor 的目的是为 Ruby 提供 Erlang 式的编程。使用 Revactor 的 pingpong 程序如图 10-36 所示。

```
1 require 'revactor'
2
3 def pingpong(name)
4   loop do
5     Actor.receive do |filter|
6       filter.when(Case[Actor, Integer]) do |partner, count|
7         if count == 0
8           puts "#{name}:done"
9           exit
10          else
11            if count % 500 < 2
12              puts "#{name}: pingpong #{count}"
13            end
14            partner << [Actor.current, count-1]
15          end
16        end
17      end
18    end
19  end
20
21 ping = Actor.spawn {pingpong("ping")}
22 pong = Actor.spawn {pingpong("pong")}
23 pong << [ping, ARGV[0].to_i]
24 Actor.reschedule
```

图 10-36 使用 Revactor 的 pingpong 程序

将图 10-36 与图 10-33 中的 Erlang 程序相比，可以看出二者构造几乎相同。Ruby 中不具备的 **receive** 语句和模式匹配，我们使用代码块或 **filter** 来实现。我认为结构上是仿造 Erlang。如果说区别，应该是以下这些吧。

- 送信不用!，而是用<<运算符。
- spawn 写成代码块。
- receive 语句换成 Actor.receive 方法。

- （Erlang 中的）模式匹配，在 Revactor 中使用 **filter** 和 **Case**（**Case** 是模式匹配的库）。
- 不用末尾递归，而是用循环。

第 24 行的发送消息后，调用 **Actor.reschedule** 方法，明确地将控制传给其他 **Actor**。本来觉得不应该有这一行，但我手头的程序，如果没有这一行，就不能正常运行。

这里虽没有详细说明，在 Revactor 中，与 Erlang 一样，错误处理也是以发送消息的方式来实现的。使用 **spawn_link** 代替（Erlang 中的）**spawn** 来生成 **Actor** 之后，**Actor** 的异常终止等通过发送消息传递给连接的 **Actor**。

Revactor 是利用 Ruby 1.9 中提供的 **Fiber** 实现的。**Fiber** 是在明确进行上下文切换时的用户级线程，有时也被称为协程（co-routine）。所以，Revactor 只有在 Ruby 1.9 上才能运行。从个人角度来讲，我期待着，像 Revactor 那样利用 **Fiber** 实现的输入输出多重化技术将成为 Ruby 1.9 的杀手级应用。

有关 Revactor 的信息，请参照 <http://revactor.org/>。Revactor 由 RubyGems 提供，用下述命令安装。

```
gem install revactor
```

但是，正如刚才所述，Revactor 只能在 Ruby 1.9 中运行，**gem** 命令也需要在 1.9 版中执行。我手头上的 1.9 版 **gem** 命令是以 **gem 1.9** 的名字安装的。

10.4.9 Revactor的应用场合

Revactor 在 Ruby 中可以像 Erlang 一样编程，其优点是可以同时体验 Ruby 的好处及 Erlang 的好处。但是，Revactor 中 **Actor** 的实现不是靠线程，而是靠 **Fiber**。Erlang 的目的在于最大限度地利用多核 CPU，而 Revactor 并不适合于这种目的。

那么，说到 **Revactor** 的最大优点，应该是能够在等待文件输入输出时，将程序的停止控制在最小程度。图 10-37 是使用 **Revactor** 进行文件输入输出的 **Echo Server**。**Echo Server** 从客户端接收 **socket** 连接请求，然后将客户端传过来的东西原封不动地返回。因为是原封不动，所以称为 **Echo**（回声）。

```
1 require 'revactor'
2
3 HOST = 'localhost'
4 PORT = 4321
5
6 # 用指定的服务器和端口生成新的接听socket
7 listener = Revactor::TCP.listen(HOST, PORT)
8 puts "Listening on # {HOST}:#{PORT}"
9
10 # 开始接受连接请求
11 loop do
12   # 接受连接请求并开始新的Actor
13   # 来处理
14   Actor.spawn(listener.accept) do |sock|
15     puts "#{sock.remote_addr}:#{sock.remote_port} connected"
16
17     # 开始回应收到的日期
18     loop do
19       begin
20         # 把读取的内容照原样输出
21         sock.write sock.read
22       rescue EOFError
23         puts "#{sock.remote_addr}:#{sock.remote_port}
24 disconnected"
25
26         # 像普通的Ruby 的socket 一样，在关闭连接时中断
27         # {并退出现在的actor}
28         break
29       end
30     end
31 end
```

图 10-37 使用 **Revactor** 的 **Echo Server**

这个 **Echo Server** 有两个重要特点。第一，与通常的 **socket** 与线程相组合的服务器相比，性能更好。特别是在同时连接数增加时，内存消

耗量和应答性能优良。这是因为与线程相比，Actor 的内存消耗量和上下文切换的成本较低。

第二，尽管性能这么优良，将程序改造为 Revactor 式，只需要很少量的修改。实际上，与通常使用 socket 和线程的 Echo Server 相比，（Revactor 式的 Echo Server）不同点仅在于：

- require 的库是 revactor，而不是 socket；
- 连接用的 socket class 是 Revactor::TCP，而不是 TCPServer；
- 个别连接处理用 Actor.spawn，而不是 Thread.new。

Ruby 中的 HTTP 服务器 Mongrel，可以用通常的线程实现，也可以用 Revactor Actor 实现。用 benchmark 对二者进行比较，不管是并行程度还是处理性能，Actor 都要好。而且，（线程式）Mongrel 服务器变为 Revactor 式，只需要几行的变更就能实现。详细请参照 Revactor 参考书。

10.4.10 另一个库Dramatis

Dramatis 是 Ruby 中另一个 Actor 库。函数型语言 Erlang 不支持面向对象功能。但是，某种程度上习惯了 Erlang 编程之后，以 process 代替对象，以发送消息代替方法调用，可以编出类似面向对象的程序。

另一方面，用 Ruby 这样的面向对象语言来实现 Actor 的时候，即使用普通对象来实现 Actor，也不可避免地会出现把方法调用和发送消息这两种类似概念混在一起的情况。Dramatis 库的目标，就是要解答这种“概念混淆”。

```
1 require 'dramatis/actor'
2
3 class PingPong
4   include Dramatis::Actor
5   def initialize name
6     @name = name
7   end
8
9   def pingpong count, partner
```

```

10     if count == 0
11         puts "#{@name}: done"
12     else
13         if count % 500 == 0 || count % 500 == 1
14             puts "#{@name}: pingpong #{count}"
15         end
16         release(partner).pingpong count-1, self
17     end
18 end
19 end
20
21 ping = PingPong.new("ping")
22 pong = PingPong.new("pong")
23
24 ping.pingpong ARGV[0].to_i, pong

```

图 10-38 使用 Dramatis 的 pingpong 程序

Dramatis 中，Actor 也是以对象方式来表现的。为了让对象称为 Actor，需要像图 10-38 中程序的第 4 行那样，装上 `Dramatics::Actor.new` 模块。已经存在的对象要变成 Actor 的话，就像下面这样：

```
Dramatics::Actor.new (obj)
```

返回一个将 `obj` 指定的对象包起来的 Actor。Dramatis 中，方法调用和发送消息没有区别。第 24 行的 `PingPong` 对象调用 `pingpong` 方法，这是通常的方法调用，执行完成之后，等待返回结果，称为同步调用。但是，同步调用不能实现 Actor 的发送消息。要实现异步调用的话，需要使用 `release` 方法。

使用 `release` 方法，可以得到异步调用的特别对象。使用这个对象的方法调用是异步执行的。因为是异步调用，返回结果之前就继续执行下面的程序，方法的返回值被扔掉。

除了 `release`，Dramatis 还另外提供等候结果的 `future` 方法。用 `release` 调用方法时，完全无视异步执行的结果，但用 `future` 可以在执行之后得到返回结果。想使用 `future` 得到执行结果时，在方

法执行完成之前，程序处于阻塞（**block**）状态，等待方法执行的完成。这是一种看起来像通常执行一样而实现了并行化的技巧。

将 **Dramatis** 和 **Erlang** 或 **Revactor** 等别的 **Actor** 处理系统比较一下，发现一个明显的特征是，方法调用和发送消息融合得非常好。与其他 **Actor** 处理系统相比，**Actor** 是对应于线程，实现并行处理的主体这一感觉虽然变淡了，但感觉到普通对象也能够作为 **Actor** 来运行，真的让人感觉很畅快。

Dramatis 不是使用 **Fiber**，而是使用 **Ruby** 的线程实现的，所以在 **Ruby 1.8** 中也能运行（**Dramatis** 也有 **Python** 版）。但是，至少在写本书时在我尝试使用的范围内，性能仍比不上 **Revactor**（而 **Revactor** 比不上 **Erlang**）。虽然如此，作为 **Actor** 的一种实现，**Dramatis** 仍然是一种很有意思的尝试，我对其前景充满期待。

* * *

以上介绍了 3 种支持 **Actor Model** 的编程技术 **Erlang**、**Revactor** 和 **Dramatis**。使用独立语言实现 **Actor** 的 **Erlang**，用 **Ruby** 提供的 **Erlang** 的功能，试图取二者之长的 **Revactor**，将 **Actor** 与对象融合而开辟 **Actor Model** 新领域（感觉是这样）的 **Dramatis**。各自思想的不同，实现特征也不同，这些都颇有意思。

我想今后的服务器端编程，能够处理大量请求的 **Actor Model** 会受人瞩目。说老实话，**Ruby** 版的 **Actor** 库的进展和性能都还差得很远，想想将来，这一条肯定是会成为受人瞩目的技术。今后我也会注意该处理系统的动向。

两个法则

一个是帕累托法则。这一法则本来是经济学中表示全体数值的大部分，是由构成全体的少部分因素所产生的，现在被广泛应用到各种各样的领域。帕累托法则通常以“80:20 法则”（80%的数值由 20%的因素产生）这种变形形式来应用。

性能优化也适用这一法则。也即程序执行时间的 80%（或者更多），都花费在 20% 的代码上。反过来说，不属于这 20%的部分，不管怎么优化，都不能给最终结果带来多大贡献。

性能优化是一个花费成本的工作，必须在所投入的成本和得到的结果之间做出平衡。

在决定投入成本（与性能）的平衡时，帕累托法则是个很有用的法则。主要的意思是，在确切测定之后，再制定改善计划。

另一个是摩尔法则，即大规模集成电路（LSI）中的晶体管数每 18 个月翻一番。这个经验法则是英特尔公司前总裁戈登·摩尔于 1965 年发表的论文中介绍的。

大规模集成电路的集成度大体上与 CPU 的性能及内存容量直接相关，40 多年来，计算机的性能几乎呈指数增长，这真是让人难以置信。半个世纪以来，计算机的进步与发展大体上遵守摩尔法则，这样说并不夸张。

但是，摩尔法则也开始被阴影所笼罩，因为快要接近物理法则的极限了。40 多年来，随着大规模集成电路集成度的提高，日常生活中意识不到的光的速度、原子大小等已经成为问题了。最近几年，有时感觉好像 CPU 时钟频率的提高处于停滞不前的状态，就是这个原因。

到现在为止，受惠于摩尔法则，CPU 的处理速度大幅变快，软件也因此而大大加速。软件开发人员什么都不用做，只要换个计算机，就自动实现了提速。

但是，幸福的时代就要结束了。单个 CPU 的提速已经接近极限。而往一个芯片内嵌入多个 CPU 的多 CPU 或者多核将成为以后的发展方向。今后，只有能够灵活利用多个 CPU 进行任务分割的软件，才能享受摩尔法则的恩惠，从而变得更加高速。

在未来高速软件的开发中，对这两个法则的认识会越来越重要吧。

第 11 章 程序安全性

11.1 程序的漏洞与攻击方法

先说明一下什么是软件的漏洞吧。其英文是 **vulnerability**，经常使用片假名的 IT 行业之所以也用汉字来表示它，是因为这个词的片假名表示发音难，且难以记忆。

所谓漏洞，是指让软件发生意外动作的可能性。软件的漏洞，根据严重程度和紧急程度的不同，可以分为多种。

其中，紧急程度低的，仅仅看做程序错误来处理就行了。但是紧急程度高的，必须作为最优先问题来解决。一听说出了安全性程序错误，有人就容易紧张得不行，事实上要分清严重程度，冷静应对。

11.1.1 4 种软件漏洞

漏洞从大的方面分为以下 4 种。

1. DoS 攻击
2. 信息泄漏
3. 权限夺取
4. 权限升格

首先，第 1 种是 DoS (**Denial of Service**) 攻击，也称拒绝服务攻击，是指妨碍软件正常运行（服务的执行）的网络攻击手段。能够引起软件异常终止的程序错误，可以说全部都是引发 DoS 攻击的安全性程序错误。

即使发生了异常终止，很多软件也都不会引起严重问题。比如说，读入损坏的数据，即使引起编辑器软件异常终止，也不会给别人带来麻烦。但是，对于有些软件，可能后果会很严重。如果读入内容有问题的邮件，而导致邮件系统整体崩溃，那问题可就严重了。

但软件即使没有漏洞，也可能从外部受到 DoS 攻击。以前，我所在的公司管理的服务器，突然有一天不响应请求了。调查后才知道，从某国集中了大量的 Web 访问，导致流量太大，处理不及。从众多 Web

浏览器同时、反复进行重新载入（reload）这种单纯攻击（Internet Explorer 的重新载入键是 F5，所以也称为 F5 攻击），使服务器负荷变得过重。结果，只要将有问题的 IP 地址进行过滤，并增强服务器功能，问题就可以解决了。这种攻击想从根本上解决是很困难的。

第 2 种信息泄漏，是指不愿公开的信息被公开了。比如用户名被公开，或者密码被看到等问题。

信息泄漏相对来说是一个重大问题，但根据泄漏信息的种类，其严重程度也是不一样的。特别是有关金钱或者个人信息被泄漏的时候，处理不慎的话，可能会发展成诉讼官司。以前也曾有过这种案例，有人因访问了不经意泄漏的个人信息而遭到起诉，说是违反了《不正当访问禁止法》。

11.1.2 因权限被窃取而成为重大问题

第 3 种权限夺取，是指夺取软件控制权，随心所欲地操纵计算机。如果权限被夺取了，计算机就成为入侵者为所欲为的工具了。也就是说，只要是在该软件所具有的权限范围以内，比如文件的写入、删除，等等，什么操作都有可能发生。如果是恶意的入侵者，就有可能破坏整个系统。

但是，软件如果只是用一般用户的权限来执行，则能够避免最恶性的伤害。一般用户所能够做的操作有限。虽说如此，但或许会有允许第 4 种权限升格这样的漏洞，所以不能过于乐观。所谓权限升格，是指夺取了一般用户权限的入侵者，获取了管理者权限。

以前，Ruby 的主页网站 www.ruby-lang.org 受到过入侵，当时是 ssh 的漏洞被利用了。虽然搞不懂夺取管理者权限的入侵者在想些什么，但他执行了

1 ssh 是 Secure Shell 的缩写，是在远程操作计算机时，通过加密通信而执行的程序。

```
rm -rf /
```

导致将所有文件递归删除的恶果（即各级目录中的文件都被删除）。估计是个犯罪狂干的。

结果，操作系统的运行所必需的文件被删除后，递归删除就停止了，但有几个文件却永远失去了。幸亏，我的重要文件都留有备份，但因为没能确定最初入侵时间，为了验证备份的文件是否被恶意改变，还是花了相当长的时间。现在回头看，这仍然是很痛心的回忆。

但是，删除所有文件这种单纯的罪行还算好的。最近有的入侵者为了不让人察觉到，自己将入侵的痕迹抹去，要么偷偷潜伏下来盗取重要信息，要么将入侵的计算机用作犯罪，真是越来越恶劣了。

11.1.3 安全问题的根源

这样的安全问题之所以会产生，而且还在不断出现，也是有原因的。

安全问题产生的根源，在于运行软件的人（权限所有者）和利用软件的人是不同的。早期的软件，利用者和软件执行权限是一致的，这种时候不会发生问题。即便软件有程序错误，程序错误的受害人也仅限于本人，说到底还是自己负责。

安全问题有 3 种情况。

1. 恶意软件
2. `setuid/setgid`
3. 服务器

第 1 种恶意软件，是指在程序本身里面植入了某种有恶意的代码。几乎所有情况下，软件的开发人员和使用者是不同的，这样怀疑起来，范围可就大了。其中，有被称为“特洛伊木马”的下了套的软件。所以，通过邮件发送来的不明就里的程序，一般都不要执行。

第 2 种 `setuid`（`set user id`），是指执行的程序以所有者权限进行动作。`setgid`（`set group id`）也用同样的方法来设定组权限。

比如，想共享游戏软件的高分记录文件，但又想避免高分记录文件被随便更改。首先把更改高分记录的程序的所有者与高分记录文件的所有者设定成一样，然后只要设定程序的 **setuid**，该程序就能以高分记录文件的所有者权限来执行。这是 UNIX 系列操作系统一直以来的做法。

这种做法的好处是，一方面根据权限对文件进行保护，另一方面根据需要可以将保护撤销。但利用软件的人会有不同的执行权限，容易成为权限升格的台阶。实际上，**setuid** 的缺点已经变得比优点更突出，现在几乎不用了。

11.1.4 “守护神”引起的问题

第 3 种是服务器，这个词本身有很多意思，这里是指为了提供服务而常驻型的软件。UNIX 中，为了将这种软件和服务器硬件本身（**server**）区别开来，称之为后台服务（**daemon**）。**daemon** 并不是恶魔（**demon**），而是鬼神或守护神的意思。

在 Linux 等 UNIX 系列操作系统中，处理邮件的 **mailer daemon**，调整时刻的 **ntp daemon** 等，有着为数众多的后台服务在运行。阿帕奇（**Apache**）的 **HTTP server** 也是一种后台服务。现在数一数，我的 Debian（GNU/Linux）机器中，有近 100 种后台服务在运行。

这些后台服务，基本上都是受理经由 **socket**² 而来的请求。执行它所提供的服务，然后将结果经由 **socket** 返回。也就是说，几乎所有的情况，利用者（发出请求者）和执行者（**daemon** 的执行权限者）都是不同的。这种软件若有了漏洞，会引起 **DOS** 问题和权限夺取问题等。

² **socket** 是 UNIX 系列操作系统中，进程间通信的一种机制。

用户彼此之间都互相认识，谁都不会有恶意的时代，早就过去了。现在，世界上充满了恶作剧者和追求一己之利的不法之徒。当然从整体上看，这些行为不端者只是少数，但绝不能无视。管理者不妨设想，给后台服务的所有输入，都是来自恶意用户。

现在的软件，很多都是通过互联网来提供。通过互联网提供的服务，基本上与通过后台服务提供的服务相同，同样有必要设想并不是所有的输入都能够信任。

正是通过互联网提供的软件，使得安全问题最近成为热议的课题。因为互联网上的软件能够简单地提供服务，门槛很低，利用者与执行权限者不同，从而很容易引起安全问题。

11.1.5 多样化的攻击手段

对软件漏洞的攻击有很多种。对软件开发者的程序错误（或者疏忽）的攻击，有代表性的有以下 5 种。

- 缓冲区溢出
- 整数溢出
- 跨站点脚本攻击（XSS）
- SQL 注入
- 跨站点伪造请求（CSRF）

并不是说这些就是全部，我想今后攻击的种类还会增加。以下简略说明这些安全攻击的内容和应对方法。

11.1.6 缓冲区溢出

缓冲区溢出是经常被提起的攻击事件。这是指向固定长的缓冲区（buffer）³ 输入了比假定的长度要长很多的数据，使程序异常终止。或者是更改堆栈的跳转地址，劫持程序。

3 为保存数据而确保的内存区域。

看看图 11-1 中的程序。这个 C 程序中使用 `gets` 函数，从标准输入读入一行，存入缓冲区（作为参数传过来的数组）。



图 11-1 有缓冲区溢出危险的程序

这个程序有一个假定，那就是“1 行的长度肯定小于 1024 字节”。确实，几乎所有的情况下，这个假定都是成立的。但并不保证对所有情况都成立。当输入者不被信任的时候，这个假定就会出卖你。当这个程序读入的输入行长度达到 1024 字节以上时，就会写入数组 **buf** 末端之后的内存（溢出），程序因而异常终止。

如果仅仅是异常终止，问题还不大。最坏的情况是程序的控制权被夺走。之所以这么说，是因为 C 语言中，局部变量是放在系统堆栈上的，往系统堆栈上以某种特定规格写入，就有可能更改函数的返回地址。

为了避免被坏人利用，这里就不详细说明了。曾经流行过一种蠕虫，通过将能够从外部进行操作的代码写入缓冲区，使 CPU 误认为已经从函数返回，在返回地址处启动 **shell**。

像这里介绍的 **gets** 一样，C 的库函数由于历史原因，虽然接受数组，但残留下一些并不进行长度检查的函数（假定数组有足够长度来容纳数据）。除了 **gets** 之外，还有 **sprintf**，**getwd** 等。现在的连接器（**linker**，将几个程序连起来的软件）很聪明，在编译带有 **gets**、**getwd** 这种有明显缺点的函数的程序时，能发出警告。这些函数有指定数组长度的“更好的替代函数”，推荐使用这些替代函数（参见表 11-1）。

表11-1 危险的C函数和替代函数

函 数 名	替代函数	功 能
gets	fgets	读入一行
getwd	getcwd	读取当前目录
strcat	strncat	字符串连接

strcpy	strncpy	字符串复制
sprintf	snprintf	字符串格式化
vsprintf	vsnprintf	字符串格式化

本来，使用 C 那种连数组长度都不检查的语言，可以说就肯定会产生问题。幸亏，像 Ruby 这样的高级语言，语言处理系统自动分配内存，可以不使用固定长的缓冲区。使用更高级的语言，可以从缓冲区溢出问题中解放出来。但由于速度上的考虑，不少人还会开发 C 语言的 CGI⁴ 及 Daemon 程序等，使用时要多加注意。

4 Common Gateway Interface（公共网关接口）的缩写，在 Web 服务器上执行程序的接口。用这个接口执行的程序，称为 CGI 程序。

11.1.7 整数溢出

整数溢出与缓冲区溢出相似，但它是更难被发现的问题。

请看图 11-2 的 C 程序。这里所示的函数，用 malloc 函数分配了一个大小为指定结构体⁵ n 倍的内存空间。

5 把程序中使用的数据集中定义在一起的结构。

```
void*
malloc_elements(size_t esize, size_t n)
{
    return malloc(esize * n);
}
```

因舍入而
变小了

图 11-2 有整数溢出问题的 C 程序

这看起来是很简单的一个函数，好像没有任何问题，但实际上却隐藏着一个很麻烦的问题。

C 等很多的语言，整数只能表示一定范围内的数。比如，无符号 32 位（二进制）整数能表示的范围是 0 到 42 亿。超过此范围，就会发生溢出，也不发警告就将数值舍入。

那么，再看一遍图 11-2 中的函数，n 虽然在 size_t 的范围之内，但当 esize * n 超越了 size_t 的范围时，就会发生问题。

假设 `size_t` 是 32 位无符号整数，`esize` 是 32。这里假设给了 `n` 一个值，134500000。

```
esize = 32
n = 134500000
esize * n = 4304000000 (本来的值)
```

这个结果超越了 32 位整数的范围，超出的位被无视。C 语言的计算结果就成为

```
esize * n = 9032704 (32 位舍入)
```

9032704 字节，也就是 9MB，对于现在的计算机来说，并不算大。这里毫无疑问，`malloc` 会分配一个 9MB 的内存空间，并将其返回。

应用程序那边本打算分配一个大得多的内存空间，结果只分配了 9MB。这样造成的结果就是，写入的数据超越了所分配的空间，最终会导致程序异常终止，发生不测。

`malloc` 不分配堆栈空间，难以发生前面所述的因缓冲区溢出而导致的权限夺取问题，但不敢断定说，绝对不会被恶用。

想要避免整数溢出，好像很麻烦。像图 11-3 所示的那样，需要注意检查计算结果是否在整数范围内。

```
void*
malloc_elements(size_t esize, size_t n)
{
    size_t len = esize * n;
    if (n != 0 && esize != len / n) {
        return NULL;
    }
    return malloc(esize * n);
}
```

检查除算后是否
返回原来的数

图 11-3 带整数溢出检查的 C 程序

顺便说一下，`malloc` 函数由于某种原因，分配内存失败的话，就返回 `NULL`。但很多程序都假定 `malloc` 不失败，肯定返回所分配的内存。这一点需要注意。

这个问题通过使用 `Ruby` 这样的高级语言可以解决。`Ruby` 的整数计算，没有 32 位溢出这一限制⁶。另外，内存分配不是由用户直接进行，内部分配都要经过严格检查。所以，只要使用 `Ruby`，与整数溢出就不沾边。

⁶ 严格来说，整数溢出发生以后，自动切换到多倍长整数去计算。

11.1.8 SQL注入

SQL 注入是对外部的输入检查不充分时所产生的典型问题。

利用关系数据库的程序，使用 `SQL` 与数据库进行交互。这时，如果稀里糊涂地将外部输入原封不动地写入 `SQL` 语句里，这样做成的程序有可能允许对数据库进行预想外的操作。

这里介绍一下漫画网站 `XKCD` 里的一个 `SQL` 注入的漫画（参见图 11-4），对白翻译成如下。

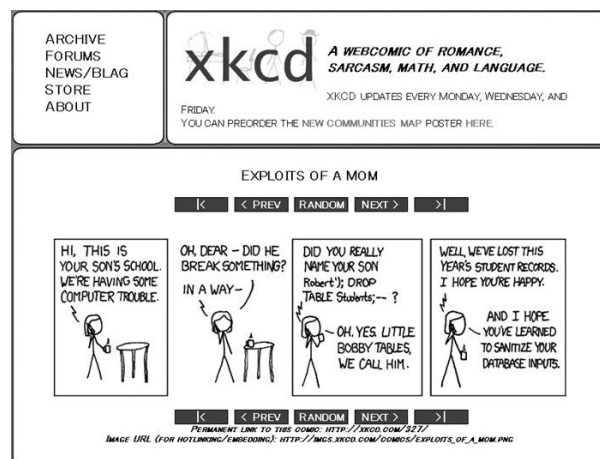


图 11-4 漫画网站 `XKCD` 登载的 `SQL` 注入的四幅漫画“妈妈的安全攻击”，URL 是 <http://xkcd.com/327/>

第 1 幅：（电话中）这里是您儿子的学校，我们的计算机出了点儿问题。（学校）

第 2 幅：天哪，他打破什么了吗？（妈妈）从某种意义上说，是。
（学校）

第 3 幅：您儿子果真叫“Robert”吗？（学校）是啊，我们叫他“LITTLE BOBBY TABLES”。（妈妈）

第 4 幅：这下可好，我们丢失了今年的学生记录。你满意了吧？（学校）

我希望你们能学会数据库的输入检查。（妈妈）

对于不知道 SQL 的人，需要额外作些说明。在插入数据的时候，SQL 用如下的 INSERT 语句。

```
INSERT INTO 表名  
('属性1' , '属性2' , ...)  
VALUES ('值1' , '值2' , ...)
```

现在，要输入 BOBBY TABLES 的数据时，单纯将字符串填进去，INSERT 语句就变成如下的样子。

```
INSERT INTO Students  
('姓' , '名' )  
VALUES ('Smith' , 'Robert');DROP TABLE Students;--')
```

本来，从 Robert 开始直到“--”为止都应当是名字，但名字中含有“'”，INSERT 语句到此结束，分号之后接着又执行了 DROP TABLE 语句（数据库删除）。“--”是 SQL 中注释的开始，剩下的“')”部分，作为注释而被忽略。用小孩子的名字对学校的系统进行了 SQL Injection，真是个超级黑客妈妈。

本来，从外部的输入不能原封不动填入到 SQL 语句中去。因为填入的文字中可能含有对 SQL 语句有某种意义的文字。

SQL 可以由外部间接赋予参数来执行的功能。比如，SQL 语句写成

```
INSERT INTO Students  
  ('姓' , '名' )  
VALUES (?, ?)
```

名和姓可以从外部输入，应该不会发生同样问题吧。

因对外部输入检查得不到位所导致的问题，不光有 SQL 注入，在 shell 和 html 中同样会发生。

11.1.9 Shell注入

Shell 注入与 SQL 注入原理相同。

```
system("ls #{input}")
```

上面的程序中，调用 shell 的时候，如果输入 input 的值是

```
data; rm -rf /
```

数据就可能会整个丢失。

从外部的输入，如果不进行检查就不能传递给 system 等危险的函数，这一点一定要特别注意。

为了从一定程度上检查出这类问题，Ruby 和 Perl 中有“污染检查”功能。给外部输入的数据加上“污染记号”，禁止对字符串进行危险的操作。

Ruby 的污染检查，是在命令行参数里附上-T 选项，或者是给程序中的全局变量\$SAFE 赋值为 1。\$SAFE 的值表示安全级别（参见表 11-2）。

表11-2 安全级别与意义

安全级别	意 义
0	外部输入的值受到了污染（默认值）
1	禁止污染值所导致的危险操作
2	禁止对进程及文件的危险操作
3	生成的全部对象都被污染
4	禁止变更全局信息

安全级别从 0（默认值）开始到 4 为止。实际利用的是 0、1、4 这 3 个值。0 是外部的输入没有信赖性问题的程序，1 是 CGI 类的外部输入需要注意的程序，4 用于不可信赖的代码。级别 4 不在我们讨论的范围。

如果将安全级别设为 1 以上，对于上述的 **system** 调用：

- **input** 是外部输入（被污染）；
- **input** 里嵌入的字符串也被污染；
- **system** 接受受污染的字符串，产生错误。

所以，在成为严重问题之前，实际已产生了错误。像 CGI 这种输入不可信赖的程序，推荐在任何时候都要将安全级别设为 1。

但是，有的 Web 应用程序框架对 **\$SAFE** 的应对不够充分，在安全级别 0 以外就不运行了。这些框架一定得改善。

11.1.10 跨站点脚本攻击

跨站脚本攻击与 SQL 注入和 Shell 注入一样，也是因为将输入值原封不动地放在输出值内而引起的问题。

比如说，做了一个 Web 公告板，如果用户输入中含有 HTML 标签（tag），就可能会出现违反公告板设置者的意图，随便嵌入图像，或者嵌入意想不到的链接之类的问题。从这个观点来说，或许应该称之为 HTML 注入。

而且，HTML 中可能夹杂着 JavaScript，这样问题就更麻烦了。因为用 JavaScript，程序能在客户端执行。如果使用 JavaScript，可以做各种恶作剧，比如：

- 弹出窗口；
- 操作画面的组成元素；
- 访问履历。

为了防止这类事情发生，将用户输入放入 HTML 之前，一定别忘了检查。

`cgi.rb` 提供了 `CGI.escape` 方法（参见图 11-5），可以用来对 HTML 进行转义处理（特殊字符的变换）。

```
<%=h req.params['input'] %>
puts CGI.escapeHTML('test<SCRIPT Language=JavaScript>alert("Hello");</SCRIPT>')
# => test<lt;SCRIPT Language=JavaScript&gt;alert(&quot;Hello&quot;);&lt;/SCRIPT&gt;
```



自动进行 HTML 转义处理

图 11-5 CGI.escapeHTML 的代码例子

Ruby on Rails 等利用的 eRuby 中，如果用 `h` 方法，可以像下述这样进行 HTML 转义（包含 `ERB::Util` 模块时——Rails 中默认包含此模块）。

很遗憾，现在的 Ruby 不会自动检测 XSS。以用户输入为基础进行输出的时候，别忘了将 HTML 标签进行转义。

把受污染的字符串原封不动地进行输出，就会产生错误，这种模板引擎⁷还没有一般化。已经有了 Tempura（URL 是 <http://www.fobj.com/tempura/>）等实施检查的模板引擎，也希望为了安全而进行的污染检查能够一般化。

⁷ Template Engine（模板引擎），一种将程序和画面剥离开发的工具。

11.1.11 跨站点伪造请求

跨站伪造请求（**CSRF**）是 **Web** 应用程序固有的攻击手段，近些年成为大家议论的话题。

几年前，社交服务网站 **Mixi**⁸ 中发生了一个事件，在用户本人不知情的情况下，日记中被写入了“你好，你好，我是玛琪！”的内容。这是利用 **CSRF** 的恶作剧。

8 **Mixi** 是日本著名的交友网站。——译者注

利用 **CSRF**，可能蒙混网络应用程序的认证，经第三者发送请求。这个事件里，由于错误地发出了第三者的写入日记的请求，导致在用户不知情的情况下往日记里写入了新的内容。

而且，如果无意间点击了这个日记中的链接的话，就会触发攻击程序，追加同样的日记内容，转眼之间，像谜一样的日记内容扩散到整个 **Mixi** 中。

这种行为的是非暂且不说，通过这个事件，以前默默无闻的 **CSRF** 名声大振。因为 **CSRF** 可以伪造各种各样的请求，潜在危害不小，所以大家开始重视它，积极研究对策，应该说是件好事。事实上，我所使用的 **Web** 应用程序中有很多原来没有 **CSRF** 对策，那件事情之后，我采取了紧急对策。

CSRF 的原理如下。

正如 **HTTP** 那一节讲述的那样，构成 **Web** 应用程序的每一页由两部分构成，一个来自网络浏览器的 **HTTP** 请求，一个是 **HTTP** 服务器的响应。本来，**HTTP** 里不含状态，为了识别网上程序一连串的交互（会话），使用了 **cookie**（**Web** 浏览器保存的信息）或者请求中的会话 **ID**。由此进行连续处理，这是一般做法。登录信息也是同样处理。

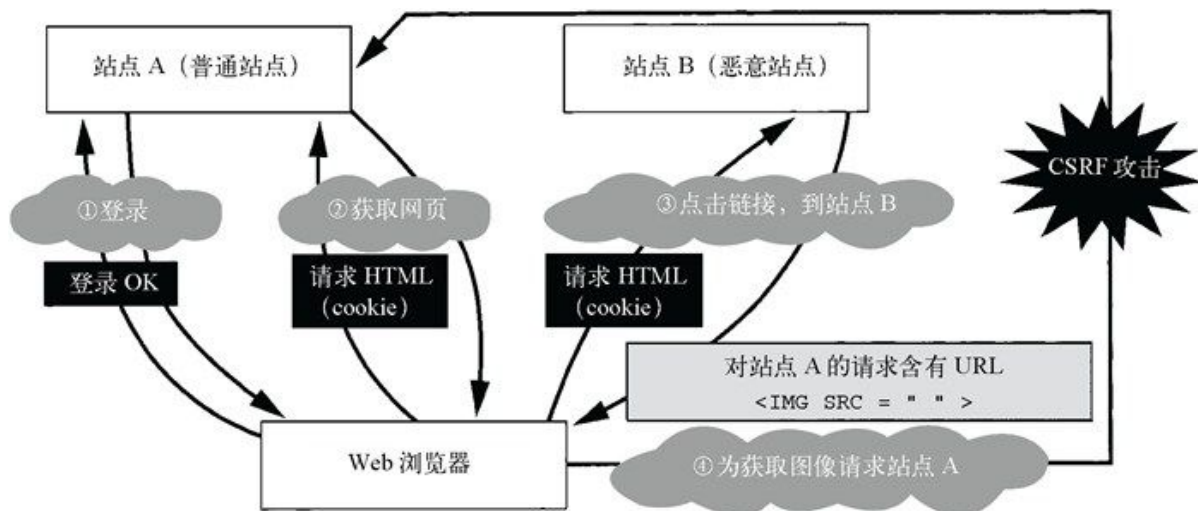


图 11-6 CSRF 的攻击步骤

假设会话 ID 的信息存放在 cookie 里，站点 A 是受攻击对象。用户正当登录到站点 A 中（参见图 11-6①）。登录成功后，站点 A 往浏览器发送一个 cookie，作为访问权的证明。以后，浏览器往站点 A 发送请求时，连带此 cookie 一起发送。站点 A 遇到此 cookie，就认为是来自正当用户的访问。

接着，用户获取站点 A 中某一页，该页含有不正当的链接。当然，该用户不知道链接是有问题的。（参见图 11-6②）

如果点击了这个有问题的链接，就会取得恶意站点 B 的某一页（参见图 11-6③）。站点 B 虽然返回 HTML，但其中含有像 这种能够自动装载的标签，标签里内嵌有能够攻击站点 A 的 URL。这就是 CSRF 的原理。

为了应对 CSRF，会话 ID 中不光要含有 cookie，还要附加能够证明这是正确请求的信息。

有 3 个方法可以实现这一目的：附加称为标记（token）的信息，检查 HTTP 请求来源（Referer）以及频繁进行认证。频繁认证检查太多了，用着不方便，一般用另外两个方法。

现在，包括 Ruby on Rails 在内，提供了某种 CSRF 对策的 Web 应用程序框架⁹ 也在增加。请参照各框架的手册。

⁹ Web 应用程序框架是指为了开发 Web 应用程序而设计的类和库。

除了会话固化（Session Fixation）之外，还有别的攻击手段。其中有很多可以由 Web 应用程序框架来对付。编写安全的 Web 应用程序的根本，就是采用有认真的安全防范措施的框架。

11.1.12 社会工程

以上讲解了由于程序的缺陷所造成的漏洞的几个例子。问题是，不光是程序的缺陷会造成漏洞，倒是更原始的方法所造成的问题更多。比如说，将密码写在小纸条上贴在显示器边框上，往客户服务中心打电话伪装成遇到困难的善意用户，问出会员信息，等等。

像这样用原始手段攻击系统的安全，有时称为社会工程攻击（social engineering）。开发人员对此无能为力，只能靠提高运用层面的防范意识来应对。

“锁链的强度取决于最弱的环节”，这是在谈到安全性时经常使用的一句谚语。虽然对策不完全靠开发人员的努力也是事实，但开发人员不能因此而放弃弥补程序上的缺陷。

* * *

这次重新思考了安全问题。安全是一个非常深奥的问题，这次的讲解只能算是接触到这个问题的皮毛。但是，很多的安全问题，可以通过语言或框架得到某种程度的解决。为了使用户不必在安全问题上烦恼，语言或者框架的开发人员必须不断努力。语言开发人员必须谨记这一点。

11.2 用异常进行错误处理

查查日语词典《广辞苑》，“异常”这个词是这么解释的：“与通常的原则不相符合，不适用一般原则，或者出现这种状况的事物。例：没有不存在异常的规则。”

具体到编程方面，“程序在执行某种处理的过程中，发生了某种异常”以及“表示这种状况的对象”被称为异常。另外，处理这种异常的功能称为异常处理功能。这种异常处理功能并不是 Ruby 所特有的，Lisp、Java、C++等多种语言都具备这种功能。

很多具有异常处理功能的语言，在发生异常以后，都会中断程序。这是因为发生了超乎预想的情况，不能指望程序再正常执行下去了。

“超乎预想的情况”，是指类似于想要打开的文件不存在这样的情况。文件不存在，也就不能从文件中读取数据，程序该如何执行也就无从知道。

这时，自动中断程序的执行，是很受欢迎的。在没有异常处理的 C 语言中，文件的打开处理估计会像图 11-7 所示的这样吧。

```
1 #include <stdio.h>
2
3 main()
4 {
5     FILE *f = fopen("/path/to/file", "r");
6
7     if (f == NULL){ /*不能打开的话，就是NULL*/
8         /*文件打开失败时的处理*/
9         fprintf(stderr, "open file %s failed");
10        exit(1);
11    }
12    ...
13 }
```

图 11-7 C 语言中打开文件

图 11-7 的程序中，打开文件的实质性的处理只有第 5 行中 **fopen** 的部分。**fopen** 函数指定文件名和打开模式，打开文件，把结果放在 **FILE** 结构体中，用于以后的读/写操作。但是，**fopen** 有时可能会失败，失败时不能生成 **FILE** 结构体，返回 **NULL** 作为错误标志。

从第 7 行到第 11 行是失败时的处理。这里表示出现错误信息之后，使用 **exit** 函数中断程序。必须一一检查有没有发生错误，这样程序变得繁杂了，而且在忘记检查时，程序可能会在前提条件不成立的情况下执行。正如上节所述，这样的检查遗漏会引发安全问题。

如果是 Ruby，程序可以写成图 11-8 所示的样子。

```
f = open("/path/to/file", "r")
...
```

图 11-8 Ruby 语言中打开文件

与 C 语言程序比较起来，简洁太多了，有点让人惊奇。只需要打开文件这一实质性的处理。如果没有特别指定，异常的状况就交给语言去处理好了。

图 11-8 中的 Ruby 程序，如果文件不存在，就会像图 11-9 的程序那样，输出错误信息，然后中断。

```
f.rb:1:in `initialize': No such file or directory - /path/to/file
(Errno::ENOENT)
    from f.rb:1:in `open'
```

图 11-9 文件不存在时的错误信息

Ruby 中发生了异常，就中断程序执行，输出该异常所对应的信息。

但是，并不是在所有情况下，都希望程序一发生异常就中断。比如在用 Ruby 写一个文字处理程序时，仅仅是在打开文件的对话框里敲错了文件名，就将整个儿文字处理程序结束，并不能说这是什么值得高兴的事。在没有明示会有什么异常时，我们希望中断程序，但在能够预测异常事态时，则希望用程序来应对。Ruby 中用 **begin** 来捕捉异常（参见图 11-10）。

```
begin                                ←异常捕捉范围的开始
  print "input filename:"           ←读入路径
  path = STDIN.gets
  path.chomp!
  open = open(path, "r")             ←打开
rescue Errno::ENOENT
  printf "No file: %s\n", path       ←文件不存在时输出错误信息
  retry                             ←从begin 开始再执行一次
end
```

图 11-10 用 begin 捕捉异常

从 **begin** 到 **rescue** 之间，如果发生了异常，那么只有在 **rescue** 里指定的异常才会被捕捉。比如在读取文件名等系统调用时发生了错误代码 **Errno::ENOENT** 以外的异常，**rescue** 中会无视，程序的执行被中断。执行的中断是从最后的调用方法开始，一直上溯查找，如果找到对应的 **rescue**，则被该 **rescue** 所捕获。如果到最后还没有找到，就显示异常信息，并终止程序执行。

Java 和 C++ 中使用 **try** 来捕捉异常，但 **try** 隐含着“试一试”的意思，我不太中意，Ruby 中就专门选了 **begin** 这个词。

虽然这样做可以让程序员集中精力作实质性的处理，带来很多很多的便利，但也并不全是好处。异常说到底还是某种形式的 **goto**，程序流程的控制变难了。但异常与 **goto** 还不同，异常没有指明发生的地方，其问题更加复杂。

那么该如何正确进行异常处理呢？后面将会讲解。

11.2.1 异常的历史

含有异常的语言中，使用人数最多的，我想应该是 Java 吧。Java 这种语言，吸收了过去的语言所提供的先进功能，并使它们普及，作出了很大贡献。但并不是 Java 发明了异常。实际上，早在 Java 流行之前几十年，异常就存在了。

最初是哪种语言开始提供异常处理功能的？很遗憾，我没能得到确切答案。我推测是 Lisp，要么就是与其相近的某种语言。Lisp 从 1972 年开始就有了异常处理功能，而 Java 是 20 年前才诞生的。

Ruby 也是在 Java 广为人知之前就具有了异常处理功能。我自己学习异常，是从一则介绍麻省理工学院（MIT）开发的 CLU 语言的报道开始的。CLU 等语言间接影响了 Ruby。

从 Java 以后，异常就变得很普通了，21 世纪的语言，几乎都具有异常处理功能。

11.2.2 Java 的受控异常

也介绍一点 Ruby 以外的语言吧。

Java 的异常与 Ruby 的异常很相似。最大的区别在于，各自的方法中是否有必要明确指出发生异常的可能性。图 11-11 显示的是 Java 的方法定义（一部分）。方法名和参数之后，用 **throws** 指定所要发生的异常。

```
void open_file() throws FileNotFoundException{  
    return new FileReader("/path/to/file");  
}
```

图 11-11 Java 的方法定义（含异常）

而且在 Java 的方法定义中，如果调用了带异常的方法，该异常既没有在异常处理中捕获，又没有 **throws** 指定，就会发生编译错误，异常也会成为方法类型的一个组成部分，这样的异常称为受控异常。被广泛使用的语言中，采用受控异常的，Java 是第一个¹。从整体而言，Java 没怎么采用新的作法，是一种保守的程序设计语言，但在这个巧妙的地方冒了一次险。

1 上述的 CLU 也声明函数产生的异常，但 CLU 怎么也算不上是“广泛使用的语言”。

为了不至于漏掉对某个异常的捕获，编辑器严格进行检查，从这个意义上说，受控异常是个难能可贵的功能。这与积极检查与静态类型不一致的类型一样，是符合 Java 方针的。

但是，也有对受控异常的批评。本来，之所以被称为异常，就是因为不好提前预想。但为了不出编译错误，编码时却要强制性地一一声明，真是件很痛苦的事。

实际上，虽然看起来与数据库文件之类的没什么关系，Java 的方法中却要抛出 **SQLException** 或 **IOException** 的例子也是有的。也就是说，让用户看到了一些并不必要的详细实现细节。

这么说来，如果削足适履，为了配合每一个方法的意义，而强制对异常进行置换，或者是为了不出编译错误，而勉强捕获异常，就有些本末倒置了。出了编译错误，也就是编译器“生气了”。如果是真的有

错，受点编译器的气那也没办法，但如果仅仅是因为异常的种类稍微有点不同就受气的话，是不利于精神健康的。而且，为了消除编译错误，必须写进大量的异常处理，异常的种种好处全被抵消了。

请不要误解。受控异常也有好处。但从个人来讲，比起那种不允许犯错的严厉老师一样的语言，我更喜欢 Ruby 这种宽容的语言。

11.2.3 Icon的面向目标判断

作为异常的变形，介绍一下 Icon 语言的目标指向判断吧。Icon 是亚利桑那大学开发的以处理字符串和模式匹配为目的的编程语言。

Icon 中的异常（Icon 中称为失败）用“伪”来表示。也就是说，计算某个值的时候没发生异常就是真，发生了异常就是伪。所以，对于 **if** 式条件判断，在 Ruby 等语言中是判断“如果式子是真”，但 Icon 不同，判断的是“如果式子成功（没发生异常）”。这样，即使是

```
a < b
```

这样简单的式子，其意义并不是“a 与 b 比较，如果 b 大就是真，如果 b 小或二者相等就是伪”，而是“a 与 b 比较，如果二者相等或者 a 大，就是异常，否则返回 b 值”。结果

```
a < b < c
```

这种式子也是正确的式子。判断这个式子的时候，**a < b** 的值在比较是真的时候是 **b**，接着再判断 **b < c**。

如果最初的比较是伪，整个式子就是失败，后面的比较就不再进行。在 Ruby 等真伪值判断的语言中，必须写成

```
a < b && b < c
```

如果条件式以外的部分判断得出失败，像其他语言中的异常一样，就中断程序执行。

这样，Ruby 中的程序段

```
begin
  # 读入1 行
  while line = gets()
    # 打印1 行
    print line
  end
rescue
  # 什么都不干
end
```

用 Icon 语言写就是

```
while write(read())
```

read 函数一行一行读取，将读取的值作为返回值传给 **write**，然后输出。文件读到最后，**read()** 函数失败。Icon 的 **while** 语句是“直到条件判断式失败为止，反复循环”的意思。**read()** 函数的失败被 **while** 语句的条件判断捕获，结束循环。

除此之外，Icon 中还有 **every** 语句，这是直到失败为止各种组合都尝试一遍的控制结构。“直到达成某种目的为止持续判断”，基于 Icon 的这种判断方式，Icon 被称为面向目标判断。

Ruby 在设计之初，也曾认真考虑过采用像 Icon 式的真伪值判断，结果还是采用了 **nil** 和 **false** 以外的值全是真值的这种正统方式。如果那时候，作出了另一种判断，或许 Ruby 的性质就会大变样。

11.2.4 Ruby 的异常

现在再来说说 Ruby 的异常吧。首先捕获异常的是 **begin** 语句。**begin** 语句的语法如图 11-12 所示。

```
begin
  可能发生异常的处理
rescue 异常类, ... =>变量
  指定的异常发生时的处理
  将异常对象赋值给变量
  省略异常类时，使用StandardError（标准错误类）
  即便省略变量，也可以用$!参照
else
  没发生异常时的处理
ensure
  跳出begin 语句以后的处理
  不管异常发生或者不发生，都要执行的处理
end
```

图 11-12 begin 语句的语法

begin 语句有 4 节。关键字 **begin** 和主体，记述可能发生异常时的处理。剩下的“**rescue**”节、“**else**”节和“**ensure**”节全部都可以省略。

跟在主体之后的 **rescue** 节，指定异常处理。Java 中称为 **catch**。**rescue** 这个名字是从 Eiffel 语言中借用的。

本来，借用的只是名字本身，实际的异常处理原理很不相同。不管怎么说，**rescue** 这个单词暗含有“将程序从异常状况中救出来”之意，不觉得很酷吗？

rescue 节指定异常类。本体在执行时，如果发生了指定的异常，**rescue** 的内容就会被执行。**rescue** 节的异常判定是以异常类为基础进行的。也就是说，发生的异常类，如果是指定的类或其子类，就认为是一致的。这对于被强调为鸭子类型，不怎么以类的层次结构为基础来处理的 Ruby 而言，是很罕见的²。

² 还有一个地方，积极利用类层次结构的是 **Numeric** 类和其子类。因为数的层次结构是从数学上来定义的。

rescue 节的类可以省略，这时候看做是指定了 **StandardError** 类。Ruby 的异常类全部继承自 **Exception** 类，如果指定了

Exception 类，那么就会捕获全部的异常。但是，后面也会讲解，我并不希望指定太大范围的异常类。

异常对象所含有的异常信息，被赋值给异常对象后紧跟着=>的变量里。这也可以省略。在省略的情况下，发生的异常可以用特殊变量\$! 引用。但\$和! 出现太多的话，会让程序看起来很丑陋。

rescue 节中，可以使用**retry** 语句。用**retry** 语句，则从**begin** 开始，包含**rescue** 节，会再执行一次。如果**rescue** 中发生异常的原因被消除了，用**retry** 再执行一遍，处理就会正常终止。但是，在**retry** 之前，如果原因没有消除，就会简单地陷入到无限循环中，这一点请注意。

对发生的每一个异常进行不同的处理，并不罕见。所以，**rescue** 节中，当然可以指定多个异常。**rescue** 节中的一致性判定从上到下执行，执行第 1 个匹配的异常。比如有多个与 **rescue** 的指定相匹配的异常发生时，实际执行的也只有第 1 个。

rescue 的后面可以放 **else** 节，这只有在 **begin** 主体中没发生异常时执行，作为成功时的后续处理。实际上 **begin** 语句的 **else** 节用途不大，从我自己的经验来看，**else** 节几乎没有必要。

放在 **begin** 语句最后的是 **ensure** 节，Java 中称为 **finally**。这个名字也是来源于 **Eiffel**。本来，**Eiffel** 中 **ensure** 并不用于异常处理，而是用于指定方法执行后应当满足的事后条件。

ensure 节里指定的处理，从 **begin** 语句跳出时肯定被执行。从 **begin** 语句的范围内跳出的方法有：

- 执行终止；
- **return** 、**break** 等；
- 异常。

不管采用哪种方法跳出 **begin** 的范围，都要执行 **ensure** 语句³。

³ 严格来讲，只有用延续（Continuation）来跳出 **begin** 的范围，才不执行 **ensure** 节。

ensure 节用于解决异常发生时，由于执行被中断所导致的不一致。关于这一点，后面还会有详细的说明。

11.2.5 异常发生

现在来看看让异常发生的方法。异常发生用 **raise** 方法。Ruby 中，**raise** 不是关键字，只是单纯的方法。调用 **raise** 方法后，生成一个异常对象，中断程序开始执行。途中，如果有与异常对象相匹配的 **rescue** 节，处理就移交给它（**rescue** 节）。在 **rescue** 中（通过 **rescue** 节的赋值，或是变量\$!），可以访问异常对象。

调用 **raise** 方法有几种形式，根据情况区别使用。首先，最基本的是仅指定错误信息。

```
raise "something bad happens"
```

这样就产生了 **RuntimeError** 异常。首先不要在意异常的种类，只要能传达错误信息，这个形式就已经足够了。

下一个形式是指定异常类与消息。

```
raise TypeError, "wrong type"
```

异常类（**TypeError**）指定 **Exception** 类的子类。**raise** 在内部生成指定类的实例，并中断程序的执行。第 2 种形式的 **raise** 中，第 3 个参数可以省略。第 3 个参数可以是数组，该数组用于回溯（**backtrace**）哪一个函数从哪一行被调用的信息。

如果想在 **rescue** 节中再现同样异常的话，需要在 **raise** 方法里指定异常对象。

```
raise exc
```

这种形式中，包括回溯在内的原来的异常信息都被原样保存，并传送给调用的上一级。

最后的形式是省略全部参数的 `raise` 方法。这种情况下，在 `rescue` 中，发生的异常保存在变量 `$!` 中。在 `rescue` 节的外侧，发生信息为空的 `RuntimeError` 异常。

11.2.6 异常类

现在来看看异常类的层次结构吧。图11-13是Ruby异常类的层次结构图。

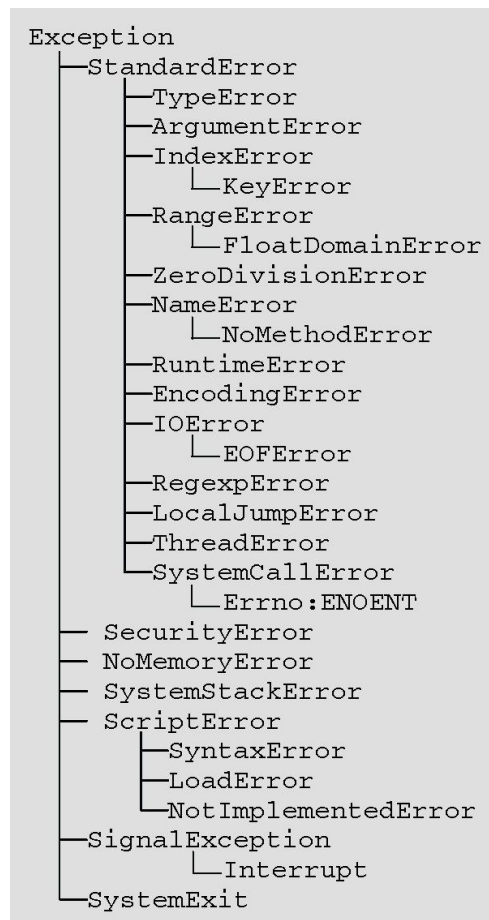


图 11-13 Ruby 的异常类层次结构图

`Exception` 是所有异常类的父类。`Exception` 子类以外的类，不能作为 `raise` 的参数。

StandardError 通常是为了表示程序执行中发生的异常事件的类。**StandardError** 是 **rescue** 节中没指定类名时的默认值。反过来说，**StandardError** 的子类以外的类需要特别的处理。

比如，**NoMemoryError** 类在内存不足、不能生成对象时发生。内存不足时能够做的事几乎没有，这个类就不能作为 **StandardError** 的子类。

ScriptError 是程序本身有错误时发生。这时比继续执行程序更优先的是修改程序错误，这个类也不作为 **StandardError** 的子类。还有，由键盘中断引起的 **Interrupt** 和 **exit** 会最终导致程序终止，此时发生的 **SystemExit** 也不是 **StandardError**。这些异常都是由错误引起的，从严格意义上说，它们不是异常（而是错误）。

操作系统的系统调用失败时，产生 **SystemCallError** 的子类异常。这些异常因 **POSIX** 规格的错误代码 **errno** 而得名。比如文件不存在时，产生 **Errno::ENOENT** 异常。这些异常类的名字不以 **Error** 为结尾，是比较罕见的个例。

11.2.7 异常处理的设计方针

如上所述，异常处理具有某些 **goto** 的特点，使用时有有必要注意一下。这里介绍一下正确的异常处理的设计方针。

首先应当考虑的是，方法的正当性。方法的执行应当“异常安全”（**Exception Safe**）。所谓异常安全，是指执行时即使发生了异常，也不会发生异常情况。比如：

- 因为发生了异常，留下了不完全的数据结构；
- 因为发生了异常，数据库里进了垃圾；
- 因为发生了异常，程序异常终止。

如果发生了上述情况，这样的方法就不能称为异常安全。异常安全的实现，说起来简单做起来难。粗心大意的话，就会在想不到的地方发生异常。为了做到异常安全，应当使用 **ensure** 来进行善后处理。

看看图 11-14。它是打开数据库进行处理的程序的一部分（大致示意）。这里用“.....”来表示数据库处理中发生了异常，程序的执行在那里中断，结果数据库没有关闭程序就结束了。

(a) 非异常安全的示例

```
db = database_open() ←打开数据库
....                ←对数据库进行处理

db.close            ←关闭数据库
```

(b) 异常安全的示例

```
db = database_open() ←打开数据库

begin                ←用begin语句将数据库处理包起来
    ....            ← 对数据库进行处理

ensure
  db.close          ← 关闭数据库
end
```

图 11-14 异常安全的程序

另一方面，在图 11-14b 中数据库处理的全体用 **begin** 语句包起来。处理中即使发生了异常，也会保证在 **ensure** 节中将数据库关闭。

虽说如此，在 **Ruby** 这样的语言中，也许一般用户对于异常安全没必要过于担心。即使有构建了半截的不完整的数据结构，或是有打开了而没有关闭的文件，某个时候也会有垃圾处理机制来回收。但是，开发数据库处理程序的底层开发人员，非常有必要意识到异常安全。

其次应当注意的是，不应当受理超出必要范围的异常，特别是应当避免在 **rescue** 节指定 **Exception**。事先完全预测会发生什么样的异常是很困难的。自己不能预测的异常，不要勉强去捕获，应当交给上层程序去处理。如果在 **rescue** 节里指定 **Exception**，就会把本来应当到达上层的没预测到的异常也给捕获了。

最后应当注意的是，除非清楚知道自己在干什么，否则处理部分不要使用空的 **rescue** 节。因为处理部分使用空的 **rescue** 节，就是无视

所发生的异常。但是，任何异常之所以发生，肯定有其发生的理由。无视异常，就等同于软件执行中没发生任何问题。这里的处理要是疏忽了，就可能错失良机，在后面酿成更大的问题。当然在有些情况下可以无视异常，但应当避免随随便便地无视异常。

11.2.8 异常发生的设计原则

那么，针对异常发生的情况，该是什么样的设计方针呢？

想要传达“与通常不同的状态”时，首先要考虑的是，如何判断该异常情况是以异常传达呢，还是以返回值传达。比如，对哈希表进行检索时，键（key）所对应的值没查到，就不返回值，这是某种意义上的异常情况。那么，这个时候，是不是该返回异常呢？当然，不存在完全正确的答案，但就我个人的想法而言，这种情况不应当返回异常⁴。

4 在 Python 中，哈希键找不到时返回异常。

容易想象到哈希表里键不存在的事，我不认为这是非得要通知调用者的重大的异常情况。除非如果不明确处理的话，程序就会异常终止，则对于这些没有办法的情况，才应该用异常。Ruby 中，对于较小的错误，习惯于返回 NIL 值，而不是返回异常。现在假设，发生的异常情况是可以作为异常的重大事件，那么是应当使用图 11-13 中 Ruby 提供的既有的异常类呢，还是制作一个应用程序专用的异常类呢？

作这个判断的基本原则是，如果想让产生的异常正好与某个既有异常类的动作一样，就使用这个既有类。这自然不会产生疑问。让人烦恼的是那些不能确信是否完全一样的情况，以及几乎相同但又想追加某些附加信息的情况。

关于前者，处理异常那一侧，应当能够判断有没有必要区分既有的相似异常和想要发生的异常。分类太细，无非是让指定变得复杂。最好是仅仅在真的想要区分时，才定义新的类。

关于后者，所谓想要追加附加信息，那肯定是预想到在什么地方会对它进行与既有类不同的处理。在这种情况下，要从既有的异常类派生一个子类，可以把对附加信息的处理交给这个子类。

在既有的异常类中，**NameError** 和 **NoMethodError** 就是这种关系。**NameError** 是在找不到指定的名称（变量或常数等）时发生的异常，而 **NoMethodError** 是在方法调用，找不到方法时发生的异常，相当于找不到名称的实体（这里是方法），从这个意义上与 **NameError** 动作相同，但只在方法调用时才存在的参数信息也想保存到异常对象中去，所以做了一个子类。

那么，假设发生的情况与既有的异常类明显不同，需要制作一个新的异常。这种情况下，必须考虑以下几点。

- 名称：应该给新的类起一个什么样的名字。
- 父类：新的类应该属于哪一个异常类的子类。
- 生成方法：应该如何初始化新的类实例。

现在进行逐一分析吧。

首先，关于名称，在异常类的末尾加 **Error**，是 **Ruby** 的习惯。如果没有特别的理由，为了明确表示该类是异常类，还是遵从习惯的好。既有的异常类中，不以 **Error** 结尾的有：

- **Exception**
- **Errno::EXXX**
- **SignalException** 与 **Interrupt**
- **SystemExit**

Exception 是所有类的根类，为了表示不能在 **rescue** 节中随便使用，没有附加 **Error**。**Errno** 模块下定义的系统调用的异常，则是为了尊重所谓 **POSIX** 错误代码的习惯，而没有附加 **Error**。

最后，关于 **SignalException**、**Interrupt** 和 **SystemExit** 这三个，虽然是利用异常这种机制，与其说是处理异常情况，不如说中断执行的性质更强一些。新制作的类，只要不是这种特殊的情况，起名字以 **Error** 结尾应该没什么问题吧。

那么，定义一个异常类吧。图 11-15 显示了生成异常类的两个方法。图 11-15a 是标准异常类的生成方法。图 11-15b 是生成一个异常对象，

然后传递给 **raise** 语句。

(a) 标准异常类定义（如果仅仅是想区别异常，这已经足够了）

```
class FooError < StandardError
end
```

(b) 附加消息以外的信息

<pre>class BarError < StandardError def initialize (mesg, info) @info = info super (mesg) end end</pre>	<p>←定义initialize 方法</p> <p>←以参数的形式追加附加信息</p> <p>←信息赋给类实例变量</p> <p>←初始化消息</p>
--	--

图 11-15 异常类定义

最后，讲解一下产生异常的两个原则。第 1 个是关于异步异常。异步异常是指由 **Thread.raise** 方法从线程外部生成的异常。语言本身虽然提供了这种功能，有这么个名字，但异步异常的基本原则是不要使用异步异常。

在预想外的时机发生的异步异常，处理起来非常困难。给异步异常编写异常安全的方法，几乎是不可能的。从我个人经验来讲，没有一次使用异步异常而不后悔的。

产生异常的第 2 个原则是文档化。为了进行合适的异常处理，什么样的方法，产生了什么样的异常，这方面的知识是不可或缺的。不同于 **Java**，**Ruby** 中没有受控异常，哪种方法可能产生哪种异常，有必要清楚详细地写成文档。

异常仅仅是对应异常情况，平常使用中不知不觉就会漏掉。但为了生产高可靠性的软件，恰好是那些平常不怎么出现的异常情况的处理，才显得尤为重要。

* * *

本节介绍了 **Ruby** 等多种语言所具有的异常处理功能。异常处理虽然非常方便，但处理异常情况有几个需要注意的地方。按照这次所学的

规则，请实施更加安全的异常处理。

安全对策的变迁

很久以前，计算机不存在安全性问题。当然，软件的程序错误以前就存在。遇到程序错误，程序虽然也异常终止，但为难的只是软件使用者本人，这算不上安全问题。

后来，计算机通过网络连接起来，安全问题一点一点地被意识到。在 20 世纪 70 年代麻省理工学院的人工智能实验室中，流行通过网络在别人的终端上玩恶作剧，直到现在还留有记录，但也只是让键盘在一段时间内不起作用，或是在画面上填满文字，虽然也有点让人讨厌，但顶多也就是闹着玩那种水平。

然而，从 80 年代开始就已经不再是恶作剧水平了。1988 年，计算机通过互联网互相连接起来，于是，莫里斯蠕虫程序也开始在互联网上蔓延开来。这种蠕虫通过攻击程序的几个程序错误而得以扩散，又因为软件设计上的一些缺陷，蠕虫以爆炸性的态势进行传播，结果计算机由于负荷过重而导致服务停止，现在这称为 DoS (Denial of Service) 攻击。

当前，几乎各种程序都有安全问题。互联网上的异常输入，从外部读取的数据被暗中捣鬼，诸如此类导致安全问题的原因，实在是太多了。日子变得不好过了。

完全消除安全问题是不可可能的，只要还有那么多问题，完全无视也是不现实的。但是，有操作系统和编程语言等底层框架的支持，我想这是否也减轻了点程序员的负担呢？Ruby 由安全级别完成数据检查的功能就是这样的尝试之一。

第 12 章 关于时间的处理

12.1 用程序处理时刻与时间

时间是我们日常生活的一部分。天亮了，睁开眼睛，吃早饭，去公司。这样的生活，时时刻刻都在带走我们的时间。但是，时间里隐藏着比我们的想象要复杂得多的因素。

12.1.1 时差与时区

在日本，当太阳公公升起来的时候，纽约却是深更半夜。国家和地域不同，此刻的时间也不相同，这称为时差。有海外旅行经验的人，肯定体验过所到的国家与日本之间的时间差异。

就是在日本国内，北海道和东京的日出时间也是有很大差别的。但随着地方的变化，时间是渐变的，这成为一件很麻烦的事。所以人们就根据国家和地区，作某种统一性的规定，划分出时区。大的国家，在国内就有时差。美国就有 6 个时区（东部、中部、山区、西部、夏威夷以及阿拉斯加）。

世界上时区的原点位置，在英国伦敦格林尼治天文台。以前，以此为基准的时间称为格林尼治标准时间（**Greenwich Mean Time, GMT**）。最近，好像“世界协调时间”（**Coordinated Universal Time, UTC**）这个叫法渐渐变成主流。

因为日本是在英国以东，所以日本时间比英国早。日本时间比 UTC 早 9 个小时，所以用+0900 表示。时区由行政区划决定，不能根据计算求得。烦琐的是，还存在与 UTC 的时间差不是整小时的时区，比如尼泊尔（+0545）和印度（+0530）。

12.1.2 世界协调时间

格林尼治标准时间与国际协调时间的区别，不仅仅是名称。格林尼治标准时间是通过观察地球旋转来计算，而国际协调时间是根据原子时钟来计算（从铯原子的振动数计算时间）。

但是，地球的旋转要受到其他天体的万有引力的影响，还有地表物体的移动，特别是潮汐等各种各样因素的影响，会产生偏差。而不受这些因素影响的原子时钟的时间，为了与实际的地球旋转相吻合，有时会插入闰秒，使这两个时间的差保持在 0.9 秒以内¹。过去 40 年间，共插入了 15 次闰秒。

1 从原理上说，删除秒也是可能的，但目前为止没有删除过秒。好像地球的自转在逐渐变慢。

说起 Coordinated Universal Time 的缩写，为什么会是 UTC 呢？大概是为了避免各语言语序的不同（英语是 CUT；法语是 TUC，Temps Universel Coordonne；意大利语是 TCU，Tempo Coordinato Universale；等等），统一采用了这种语序。

12.1.3 夏令时 (DST)

还有，成为问题的夏令时。这是因为到了夏天，日照时间变长，太阳出来了还在睡，太浪费。将时间错开以有效利用珍贵的日照时间，夏令时就是基于这种想法。呼吁要节省能源的今天，日本也要导入夏令时的法案屡屡成为话题。

日本称夏令时，英语中一般称为 Daylight Saving Time (DST)。夏令时虽然由法律规定，但从何日开始到何日为止适用夏令时，因国家不同而不同。比如美国是从 3 月的第 2 个星期日开始到 11 月的第 1 个星期为止适用夏令时。

虽说是“夏令时”，但却不像大家想的那样只适用夏季。在美国，2006 年以前，夏令时是 4~10 月，现在又延长了，已经超过半年了。让人觉得，夏令时反而成了主流。

基本规则说起来很简单，一定期间内时钟拨快一小时。但实际编程的时候，时间切换就成了问题。

导入了夏令时，春秋两回该如何调整时钟，实际来看一看吧（参见图 12-1）。作为例子，图 12-1 显示了 2008 年美国东部时间 (EST) 与美国东部夏令时 (EDT) 之间的切换。

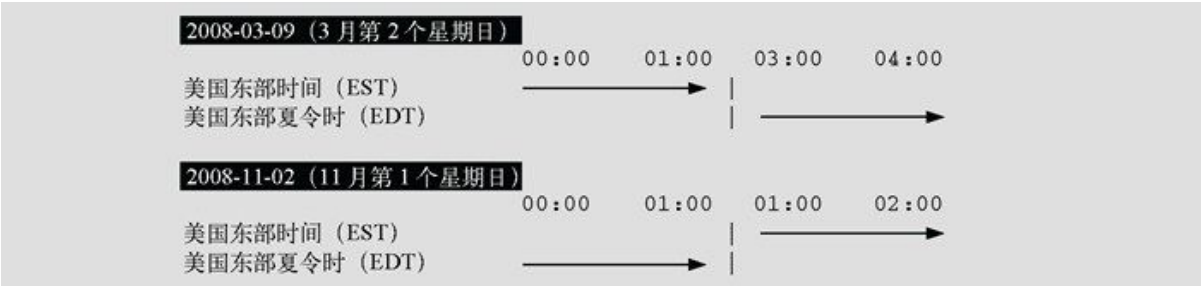


图 12-1 夏令时的开始与结束

夏令时开始那一天，凌晨 2 点没有了。标准时间从凌晨 1 点 59 分直接跳到 3 点，时钟就早了一个小时。

反之，结束那一天，夏令时的凌晨 1 点的下一小时是标准时间的凌晨 1 点，这样时钟就慢了一个小时。美国的夏令时切换时间是凌晨 2 点，但各个国家的法律规定不一样。规定凌晨 1 点的国家也有（欧洲各国），规定午夜 0 点的国家也有（巴西）。

这可以亲眼确认一下。Linux 中，通过一个环境变量来设定时区，将环境变量 **TZ** 设置为表示时区的字符串就行了。将时区强制设定为美国东部时间吧。

```
% export TZ=US/Eastern
```

这样就成了美国东部时间（纽约等地）。

在这种状态下运行一个简单的程序（参见图 12-2），可以体验一下时区的移动。

```
# 2008-03-09 (3 月第 2 个星期日)

[[2008,3,23],[2008,11,2]].each do |date|
  t = Time.local(*date)
  4.times do
    p t
    # 显示 1 小时后的时刻
    t += 60 * 60
  end
  puts
end
```

执行结果

```
Sun Mar 23 00:00:00 +0900 2008
Sun Mar 23 01:00:00 +0900 2008
Sun Mar 23 02:00:00 +0900 2008
Sun Mar 23 03:00:00 +0900 2008

Sun Nov 02 00:00:00 +0900 2008
Sun Nov 02 01:00:00 +0900 2008
Sun Nov 02 02:00:00 +0900 2008
Sun Nov 02 03:00:00 +0900 2008
```

12-2 夏令时移行显示程序

我个人经历过 2007 年 11 月在美国北卡罗来纳州举行的 RubyConf (Ruby 会议) 的最后一天和 2008 年 3 月在捷克的布拉格举行的 EuRuKo 的第二天，正好都是夏令时的切换时间。两个会议的主持人在前一天，都反复强调“明天切换夏令时了，不要搞错时间”。即便如此，忘记调整手表时间的人还是比比皆是。旅馆房间的时钟当然不会自动调整，除非特别注意，而一不留神犯错误的人一个接一个。如果是电波时钟，还会自动调整。但把全国所有的时钟一下子全弄成电波时钟也是不可能的。

美国大多数的州和欧洲大多数的国家，都比日本纬度高，冬天与夏天的日照时间差别相当大。在这样的地域，实行夏令时还是有它的好处的。像日本这种低纬度国家，实行夏令时并没带来什么利益，反而引起生活的变化和程序的修改，全是成本。实际上，实行夏令时的都以美国和欧洲为主，日本周边的中国、韩国等也没有实行。

说起来也是很久以前的事了，Ruby 1.4.4 的时候（2000 年左右），处理夏令时曾经有过一个程序错误。那个时候，Ruby 总算开始有欧洲用户不断加入，从他们的一个报告里，我发觉了这个程序错误。这是一

个典型的边界条件错误，夏令时刚开始和刚结束的几个小时，时刻错了。由于日本没有夏令时，我自己不能理解问题，所以修正这个程序错误花费了好大劲儿。从这次的教训就可以知道，轻易地导入夏令时，会导致软件问题频发，最坏的情况会成为社会问题，即便不出现最坏情况，也会让众多默默无闻的程序员忍气吞声，贸然实行夏令时的做法，我实在是不敢苟同。

日本国内本来没有时差，也没有夏令时，在时间运营方面是一个幸福的国家，特意导入夏令时，是不是有意让事态复杂化呢？节省能源应当采用别的手段吧。

12.1.4 改历

我们使用的日历，是格里高利历（中国称公历）。4 的倍数的年² 是比平年多一天的闰年。依靠这个来补正地球公转周期和日历的偏差。但是，格里高利历并不是自古以来就使用的，过去曾使用尤利乌斯历。在欧洲（及其殖民地），过去几百年间，分别从尤利乌斯历（旧历）切换到了格里高利历（新历），这称为改历。

² 正确来讲，是 4 的倍数，“除掉不是 400 的倍数而只是 100 的倍数”的年。所以，2008 年是闰年（4 的倍数），但 2100 年（100 的倍数）不是闰年，2000 年（400 的倍数）是闰年。

比如英国于 1752 年 9 月改历。UNIX 的 `cal` 命令对应着英国的改历，实际看一看吧。用 `cal` 命令显示 1752 年 9 月就能知道，从 2 日到 14 日之间的日期有跳跃。这是由改历，也就是日历切换所造成的日期跳跃。尤利乌斯历和格里高利历差了那么多。

主要国家的日历切换如表 12-1 所示。考虑到最早的是意大利从 16 世纪开始改历，到了 20 世纪还有没改历的国家，这种对比真让人惊异。

表12-1 改历的日期

国 名	改历的日期
德国	1700年03月01日
西班牙	1582年10月15日
法国	1582年12月20日
英国	1752年09月14日

希腊	1924年03月23日
意大利	1582年10月15日
俄罗斯	1918年02月14日
美国	1752年09月14日

12.1.5 日期与时间的类

Ruby中，表示日期与时间的类有很多。像下面这样，各自的功能及限制均有不同，有必要按目的分别使用。

Time 类

表示日常所用时间的类。是用 C 实现的内嵌类，使用与 POSIX 的时间相关的 API 来实现。虽然也对应时区，但一个程序中，只能使用本地时间和国际协调时间（UTC）两个时区。能够表示的时间范围有限制（1970 年 1 月 1 日到 2038 年 1 月 19 日）。

Date 类

表示不含时刻的日期的类。使用时，需要将 date 库加载（require）进来。与 Time 类不同，其特征是能够表示的范围事实上没有限制。改历也有对应。

DateTime 类

Date 类附加上时间信息的类。能表示时间，而且没有范围限制，功能上最强，但因为是用 Ruby 实现的，所以其性能有些让人担忧。DateTime 类说到底还是在 Date 类里加入时刻信息，与 Time 类的方法没有互换性，这一点需要注意。利用 DateTime 类，需要将 date 库加载进来。

如果是日常使用就用 Time 类，要处理遥远的过去及未来的时刻，用 DateTime 类比较好。

Time 类

Time 类的概要如表 12-2 所示。Ruby 的 **Time** 对象，以 POSIX 的时间函数为基础而设计。POSIX 的时间函数具有以下特征。

表 12-2 Time 类的主要方法

类方法	
<code>Time.at(time[, usec])</code>	从经过的秒数生成 Time 对象
<code>Time.gm(year[, mon, day, hour, min, sec])</code> <code>Time.utc(year[, mon, day, hout, min, sec])</code>	生成 Time 对象（世界协调时间）
<code>Time.local(year[, mon, day, hout, min, sec, usec])</code> <code>Time.mktime(year[, mon, day, hout, min, sec, usec])</code>	生成 Time 对象（当地时间）
<code>Time.now</code>	生成相当于现在时刻的 Time 对象
<code>Time.parse(str[, now])</code>	解析字符串，生成一个 Time 对象。需要将 <code>time</code> 库加载进来
<code>Time.iso8601(str)</code> <code>Time.jisx0301(str)</code> <code>Time.rfc822(str)</code> <code>Time.rfc2822(str)</code> <code>Time.rfc3329(str)</code> <code>Time.xmlschema(str)</code>	依据特定的格式，从字符串生成 Time 对象。需要将 <code>time</code> 库加载进来
<code>time + n</code>	返回 <code>n</code> 秒后的 Time 对象
<code>time - time</code>	以秒数返回两个时刻间的差（ Float ）
<code>time - n</code>	返回 <code>n</code> 秒前的 Time 对象
<code>time > time</code> <code>time >= time</code> <code>time < time</code> <code>time <= time</code> <code>time <=> time</code>	比较时刻
<code>time.asctime</code> <code>time.ctime</code>	返回表示时刻的字符串
<code>time.localtime</code>	以后用当地时间来表示时刻
<code>time.gmtime</code> <code>time.utc</code>	以后用世界协调时间来表示时刻
<code>time.gmt?</code> <code>time.utc?</code>	时区设定为世界协调时间时，返回真
<code>time.getlocal</code>	返回设定为当地时间的新的 Time 对象
<code>time.getgm</code> <code>time.getutc</code>	回设定为世界协调时间的新的 Time 对象
<code>time.gmt_offset</code> <code>time.utc_offset</code>	以秒为单位返回与世界协调时间的时差

time.gmtimeoff	
time.sec time.min time.hour time.mday time.day time.mon time.month time.yday time.year time.zone	返回time 的各自部分
time.isdst	如time 是夏令时，返回真
time.strftime(fmt)	按格式表示时刻
time.to_f	以秒为单位返回time （浮点数）
time.to_i	以秒为单位返回time （整数）
time.tv_sec	以整数返回自epoch 起经过的秒数
time.tv_usec time.usec	返回时刻的微秒部分
time.tv_nsec time.nsec	返回时刻的纳秒部分[1.9]
time.monday? time.tuesday? time.wednesday? time.thursday? time.friday? time.saturday? time.sunday?	time 是一周中的该日时，返回真[1.9]

- 以自某一固定时间 epoch（世界协调时间 1970 年 1 月 1 日凌晨 0 时）起经过的秒数表示时刻。每次以此为基础计算日期（年月日等）。
- 不管是世界协调时间还是当地时间（local time），都可以处理。当地时间的时区可以由环境变量来设定，但一个进程中不能在多个时区中切换。

Ruby及UNIX的时刻模型能够处理当地时间和世界协调时间。反过来说，处于这两个时区以外就不能简单处理。就像图12-3中这种感觉。

```
t1 = Time.local(2008,08,08,12)
p t1
# 结果 => Fri Aug 08 12:00:00 +0900 2008

2008 年08 月08 日正午 (UTC)
t2 = Time.gm(2008,08,08,12)
p t2
# 结果 => Fri Aug 08 12:00:00 UTC 2008
```

图 12-3 执行 Time 的 local/gm 方法

Time 对象可以设定是以当地时间表示时刻，还是以世界协调时间表示（参见图 12-4）。

```
p t1
# 结果 => Fri Aug 08 12:00:00 +0900 2008

设定为国际协调时间
t1.gmtime
p t1
# 结果 => Fri Aug 08 03:00:00 UTC 2008

回到本地时间设定

t1.localtime
p t1
# 结果 => Fri Aug 08 12:00:00 +0900 2008
```

图 12-4 gmtime 以及 localtime 方法的执行示例

Linux 的时刻函数是引用环境变量 TZ 来决定当地时间的，所以程序通过切换环境变量，也可以勉强使用多个时区。

```
original_tz = ENV["TZ"]
ENV["TZ"] = "US/Eastern"
p Time.now      #东部时间
ENV["TZ"] = original_tz
p Time.now      #当地时间
```

但是，切换环境变量的副作用很大（比如说线程会有问题），不推荐使用。

Time 类的方法中，含 **gm** 或 **gmt** 的太多了，都是来自世界协调时间使用之前的格林尼治标准时间。**Time** 类的方法名是以 **UNIX** 系列时间函数为基础的，让人浮想起，原来 **UNIX** 的时间函数，在确定 **UTC** 名称以前就有了。**Ruby** 中追加了来自 **UTC** 的方法别名。

Ruby 的 **Time** 对象，将 **POSIX** 的时刻函数与结构体整齐地归结为类，非常便于使用。不是将整数值解释为时刻，而是“时刻就是时刻”的处理方式，这是很可贵的。最难能可贵的是，**Ruby** 全体在表示时刻的部分中都用 **Time** 对象。所以，

```
p File.mtime("ChangeLog")
# => Tue Jul 08 15:04:07 +0900 2008
```

能明确地表示成时间。如果像“1215497047”这样表示成秒数的话，直觉上看不出这是什么时候吧。

12.1.6 2038 年问题

虽然 **Time** 类是如此方便的一个好类，但关于时间，要照顾到方方面面，还是有弱点的。说来说去，最大的弱点还是它能够表示的时间范围有限。

不仅限于 **UNIX**，很多的操作系统中，都是以过去某个时点开始所经过时间来表示时刻的。与人们习惯的以年月日的组合来表示时刻形成对照。在 **UNIX** 中，过去某个时点是指 1970 年 1 月 1 日凌晨 0 时（**UTC**）。

比如 **Ruby** 的誕生日（1993 年 2 月 24 日）的日本时间中午，计算机中以 730522800 这个数字来表示。**Ruby** 中可以用以下代码确认。

```
p Time.local(1993, 2, 24, 12, 00).to_i
# => 730522800
```

问题是，计算机能够处理的整数，大小有限制。如果系统的整数是 32 位（二进制）带符号整数的话，能够表示的最大整数是 2147483647，最小整数是 -2147483648。虽然看上去是 21 亿多这么一个挺大的数，实际上却是连全人类的人口数都不能表示的很小的数。如果用秒数来数这么一个并不很大的数，从刚才所说的 1970 年 1 月 1 日凌晨 0 时开始数，界限是在 2038 年 1 月 19 日（星期二）3 时 14 分 7 秒（UTC）。

结果，32 位系统中的 **Time** 类的对象，只能表示 1970 年 1 月 1 日到 2038 年 1 月 19 日的范围³。也就是说，1970 年以前出生的人连自己的生日都不能表示。

³ 这是秒数是正数的情况。POSIX 标准中虽然没有明示，使用负秒数的平台也很多，包括 Linux 在内的这种平台，过去的时间可以表示到 1902 年 12 月。

过去当然是一个重要问题，但未来的问题则更为严重。完全不能处理 2038 年 1 月以后的时间。离 2038 年虽然还有 30 年，但在银行贷款的计算等情况下，30 年以后的时间已经现实性地慢慢逼近了。如果完全不能进行这些计算的话，情况就不乐观了。类比 1999 年左右闹得沸沸扬扬的“2000 年问题”，这个问题有时被称作“2038 年问题”。

到那时候，是不是所有操作系统都变成 64 位了？希望是那样。如果表示时刻的整数变成 64 位了，问题就会延后很久很久。到公元 292277026596 年 12 月 4 日（星期日）15 时 30 分 7 秒（UTC）为止都不会有问题。这个未来时间已经用图 12-5 的程序确认了。

```
require 'date'
last = (1<<63)-1
puts Date.new3(1970,1,1)+(last/(24*60*60))
# => 292277026596-12-04
p (last%(24*60*60)/3600)
# => 15
p (last%(24*60*60)%3600/60)
# => 30
p (last%(24*60*60)%3600%60)
# => 7
```

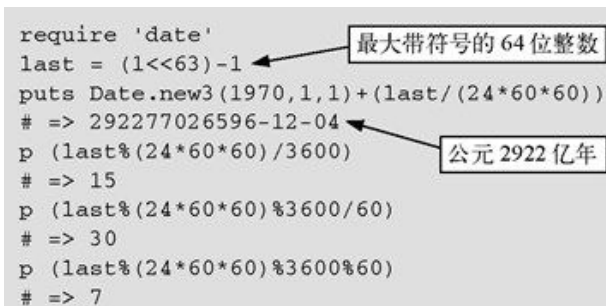


图 12-5 64 位时间的界限

也只有没有限制整数大小的 **Ruby** 才能这样轻松地计算。这个暂且不说，公元 2922 亿年，有点超越想象了。希望 2038 年之前计算机系统都能够转变成 64 位。

12.1.7 DateTime 类

相对于以epoch（某个时点）开始的秒数来管理的Time 类，DateTime 类是以日期为基础计算的 Date 类，附加上时刻信息而生成的。Date 类以日为单位来表示（光凭这一点就有 $24 \times 60 \times 60 = 86400$ 倍的分解能力），而且用Ruby的多倍长整数，事实上能够处理无限的范围。与 Time 类比起来，DateTime 的不同点在于API与时区。

首先是 API，DateTime 类是表示日期的 Date 类附加上时刻信息。啰里啰嗦再重复一遍，与 Time 类没有互换性。

其次是时区，DateTime 类虽然理解与 UTC 的时差，但没有时区的概念。做一个新的 DateTime 对象，默认值是生成一个与现在 UTC 有时差的当地时间的 DateTime 对象。

使用 DateTime 类，需要将 date 库加载进来。DateTime 类的概要示于表 12-3 中。

表 12-3 DateTime 类的主要方法

类方法	
DateTime.civil([year,mon,mday,hour,min,sec,offset,start]) DateTime.new([year,mon,mday,hour,min,sec,offset,start])	生成相当于日历日期的DateTime 对象
DateTime.commercial([cwyar,cweek,cwday,hour,min,sec,offset,start])	生成相当于商业日期的DateTime 对象
DateTime.jd([jd,hour,min,sec,offset,start])	生成相当于尤利乌斯日期的DateTime 对象
DateTime.now	生成相当于现在时刻的DateTime 对象
DateTime.ordinal([year,mon,mday,hour,min,sec,offset,start])	生成相当于年日期的DateTime 对象
DateTime.parse(str)	解析字符串，生成DateTime 对象
DateTime.strptime(str,fmt)	按指定格式从字符串生成DateTime 对象
DateTime.iso8601(str) DateTime.jisx0301(str)	按指定格式从字符串生成DateTime 对

DateTime.rfc822(str) DateTime.rfc2822(str) DateTime.rfc3339(str) DateTime.xmlschema(str)	象，需要将 date/format 库加 载进来
实例方法	
datetime.sec datetime.min datetime.hour datetime.mday datetime.day datetime.mon datetime.month datetime.yday datetime.year datetime.zone	返回datetime 的该 部分
datetime.sec_fraction	以有理数返回1秒 以下的单位
datetime.new_offset(offset)	返回设定了新时差 的DateTime 对象。 offset 指定为有 理数（1小时作为 1/24）或+0900这 样的字符串
datetime.offset	返回与UTC的时差
datetime.strftime	按特定格式显示时 刻
datetime.zone	返回表示时区的字 符串
datetime.iso8601 datetime.jisx0301 datetime.rfc3339 datetime.xmlschema	返回将datetime 格 式化后的字符串， 需要将 date/format 库加 加载进来

作为使用 **Date** 类的例子，有一个计算天数的程序。图 12-6 是一个计算从 **Ruby** 诞生以来经过了多少天的程序。

```
require 'date'

# 今天Ruby 诞生多少天了？
ruby = DateTime.new(1993,2,24)
today = DateTime.now
printf "今天Ruby 诞生%d 天\n",(today - ruby).to_i
```

图 12-6 计算 Ruby 诞生以来所经历时间的程序

输出类似于“今天是 Ruby 诞生的第 5612 天”。

12.1.8 Time 与 DateTime 的相互变换

`Time` 与 `DateTime` 很相似，但 API 不同，不能使用 Duck Typing 互相替换。

Ruby 1.9 中，`Time` 类与 `DateTime` 类分别追加了 `to_time` 方法和 `to_datetime` 方法，使用这些方法，可以把对象的类统一起来。另外，这两个方法在 Ruby on Rails 的 ActiveSupport 库里也提供，Rails 用户不用等 1.9 版本就可以使用。

* * *

时间和日历虽然大家每天都接触，但实际编起程序来，却有各种各样不为人知的细节。Ruby 中，有了 `Time` 类和 `DateTime` 类，不必太钻研这些细节就能够操作时间。

时间的难点

几乎所有人都认为时间很简单，从过去到未来，永不停息一直在流动。如果不考虑宇宙大爆炸之前有没有时间这些疑难问题，这个认识基本是事实。

但实际上深入思考一下时间，会意外地发现有很多疑难问题潜伏在身边。

首先介绍“想当然”的问题。不久以前有“2000年问题”，这是由于以公元纪年的后两位来表示年，到了21世纪，数字反而变小了。也许当时的软件开发人员没想到那时开发的程序居然能用到21世纪吧。21世纪其实并没有想象中那么遥远。说实话，本来以为21世纪会更加发达一些呢，可现在连铁臂阿童木都还没有诞生的迹象呢。

对于时间的另一个想当然的问题，是“2038年问题”。正文中已经介绍过了，为了表示时间，使用从某一原点（1970年1月1日）开始的秒数，结果令人意外地很早就到达了 32 位整数的极限。这也是由于“想当然”而造成的问题。

还有，日历也是一个麻烦的问题。现在我们使用的历法，是称为格里高利历的太阳历，为了补正地球的自转时间和公转时间的偏差，4 年 1 次搞一个 366 天的闰年，但这又有一点补正过头了，于是 100 年里又有一次 4 的倍数不是闰年的年，这还没完，（100 的倍数中）400 年里还是有一次闰年。真是太麻烦了。而且，世界上并不是全都使用同一种历法。伊斯兰国家直到现在，还在使用以月亮的圆缺为基准的太阴历。历史上，历法被政治所利用的事件很多，古罗马皇帝为了让以自己姓名命名的两个月（July 与 August）更突出，让本来意思是 8 月的 October 挪到了 10 月。

时间还有时差的问题。我们曾试图让 Ruby 的开发人员集中起来，开一次网上会议，但因为大家分别住在日本、美国以及欧洲的不同地方，确定一个共同的时间都很费劲。海外旅行时，往家里打个电话也很犹豫。

仔细想想这些，就会发现人类的时间标记系统真是奇妙。1 天用 2 组 12 个小时表示，1 小时有 60 分，1 分有 60 秒，二进制与 60 进制混在一起。干脆一下子将一天分成 1000 份，也省去了这么多麻烦的计算。

没想到还真有人这么决定了。那就是瑞士的斯沃琪公司所提倡的新的时间表示法“斯沃琪互联网时间”。将 1/1000 天定为一个单位，称为比托。1 比托相当于 1 分 26.4 秒。时间用 @517 这样的方式来表示。这表示一天开始后，过了 42600.8 秒。互联网时间没有时差，一天的开始由斯沃琪公司所在地的瑞士时间来决定。

这真是一个大胆的时间系统，但也许太大胆了，很难扎下根来让大家接受。

第 13 章 关于数据的持久化

13.1 持久化数据的方法

Ruby 程序中的对象，与现实世界中的“物”不同，只存在于计算机的内存中。所以，一旦程序的执行完成了，内存就会被回收，对象也随之灰飞烟灭。这里，就需要把必要的数据保存在文件里，以便下次还可以再读出来。

但是，还有这么一个世界，数据不随进程一起消失。比如，应该称为面向对象鼻祖的 **Smalltalk** 程序。**Smalltalk** 中，每次执行结束时，程序的执行状态都被保存在称为“映像”的文件里，下次执行时，能够恢复成与上次完全相同的对象状态。

这样的话，保存到文件的概念就没什么必要了，只要做一个普通的对象，然后就原封不动地“持久化”。也就是说，所有的对象都具有超越进程的寿命。这是一个理想的世界，但反过来讲，则是做了一个封闭于 **Smalltalk** 的世界。如果系统全都是由 **Smalltalk** 构成的倒还好，但如果想与其他系统协作的话，就会有许许多多的麻烦。

我们住在使用 **Ruby** 的世界里，这个世界与 **Smalltalk** 不同，还是有必要将数据（对象）保存到文件里。“将数据保存到文件里”，这种说法很生硬，于是就套用一个说法，称之为“持久化”。

13.1.1 保存文本

如果保存的数据是文本的话，问题就简单了。**Linux** 等 **UNIX** 系列的操作系统中，能够用文本表示的东西（字符串）可以简单地放到文件里。**Ruby** 的 **IO** 类（及其子类 **File** 类）是实现文本输入输出功能的类，比如将 **hello world** 字符串写入到文件，再读出的程序示于图 13-1。

```
data = "hello world\n"
#往当前目录的data 文件里写入
open("./data", "w"){|f|
  f.write data
}
#从data 文件中读出
open("./data", "r"){|f|
  data = f.read
}
```



图 13-1 文本输入输出程序的示例

本来文本处理就是 Ruby 这种脚本语言的主要目的之一，所以 Ruby 对这样的处理很拿手。

13.1.2 变换成文本的Marshal

那么，如果写入或读取的数据不是单纯的字符串，而是数组或对象的情况，那么该怎么办呢？

正如刚才所说的，UNIX 系列操作系统中，文件可以看做是文本数据的容器。Windows 中也一样。就是说，将对象按一定的方式变换为文本，就可以保存到文件中去。这样的对象文本化，就称为 **serialize**（序列化），或是 **marshal**（封送处理）。

根据字典，**serialize** 是序列化，按顺序排列的意思，表示将对象置换为字节的排列。另外，**marshal** 意思是将军队整列，与 **serialize** 意思是一样的。

Java 中常使用 **serialize** 这个词，而 Ruby 中多称为 **marshal**。**serialize** 这个词在并行处理中用于完全不同的意思。为了不至于引起误解，这里使用 **marshal** 这个词吧。

marshal 比表面上看起来要麻烦。对象一般含有对其他对象的引用。由于引用，多个对象连接起来，有时多个对象引用一个对象（参见图 13-2）。

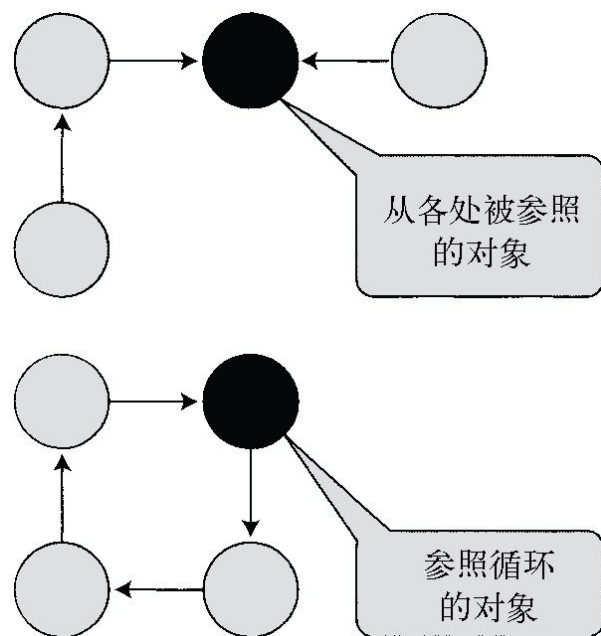


图 13-2 对象的 marshal 较难的例子

单纯将对象的引用都保存起来的办法，会引起多次引用的对象被保存多次的问题，更有甚者，在循环引用的时候，有可能陷于无限循环。

13.1.3 使用Marshal模块

标准 Ruby 中，嵌入了 marshal 功能，这就是 Marshal 模块。

Marshal 模块中，提供了几乎能将全部 Ruby 对象变为字节串的方法 `dump`，以及将字节串恢复成原对象（的复制）的 `load` 方法（参见表 13-1）。

表13-1 MiniDB 的类方法

方 法 名	内 容
<code>dump(object [,io][,limit])</code>	将object 与它所引用的对象变换为字节串
<code>load(from [,proc])</code>	以dump 变换而得的字节串（字符串），或保存这种字节串的io 对象作为参数，返回与原来（dump 前）的对象状态相同的对象

简单说明一下 `dump` 和 `load` 的使用方法吧。

dump 的第 2 个参数指定为 **IO** 对象时，就把变换结果写到该 **IO** 对象。否则，返回变换后的字节串。如果指定了整数 **limit**，那么在逐层深入处理被引用的对象时，遇到对象的连锁深度比 **limit** 还大的情况，就产生异常。

load 的第 2 个参数指定为处理对象 **proc** 的时候，对复原的各个对象都要调用 **proc**。

使用 **Marshal**，可以把复杂结构的数据简单地写入文件。当然，像图 13-2 中所举的例子那样，同一个对象多次引用，以及循环引用的情况也都能应对。

因为 **Marshal** 知道基本的对象构造，所以 **Marshal** 模块的 **dump** 方法会自动完成从普通对象到字节串的变换。像图 13-3a 那样，使用起来特别简单。

这里 **obj** 对象以及它所引用的全部对象被变换为文本，赋值给 **data**。直接输出到文件时，程序如图 13-3b 所示。这种情况下，变换结果不存入内存而是直接写入文件，效率有明显提高。

读取 **dump** 转储下来的数据，程序如图 13-3c 所示。**obj2** 被赋值为原对象的一个副本。从文件读取时，程序如图 13-3d 所示。

```
(a) 变换成字符串
data = Marshal.dump(obj)

(b) 输出到文件
f = open("/tmp/data", "w")
Marshal.dump(obj, f)

(c) 读取
obj2 = Marshal.load(data)

(d) 从文件中读取
f = open("/tmp/data", "r")
obj2 = Marshal.load(f)

(e) 深复制
obj2 = Marshal.load(Marshal.dump(obj))
```

图 13-3 Marshal 的使用方法

像这样使用 Marshal，对象可以简单地保存到文件里。

13.1.4 复制有两种方式

使用 Marshal，可以完成对象的深复制（deep copy）。

复制对象的时候，通常使用`clone`方法。这种情况下，只复制直接对象，引用的对象不复制。这称为浅复制（shallow copy）。

而深复制连同引用对象也一起进行递归复制。使用 Marshal 可以像图 13-3e 那样实现深复制。

13.1.5 仔细看Marshal的格式

Marshal 用二进制形式将对象文本化。所以，并不能直接看明白是什么意思。但想要看转储下来的数据内容的情况也不很多，因而问题也不大。为了满足那些想看的人，将 Marshal 的数据格式列于图 13-4 与表 13-2 中。图 13-5 显示了一个 Marshal 数据的例子，Marshal 就输出这样的二进制数据。

[major][minor][object]...

major:文件格式主版本（Ruby 1.8.6 中是4）

minor:文件格式次版本（Ruby 1.8.6 中是8）

object:[类型][类型固有表示]

图 13-4 Marshal 数据的格式，从先头开始，依次为 major、minor、object

表13-2 构成Marshal数据的类型一览

类型	内容
0	nil
T	TRUE
F	FALSE
o	对象、类实例变量、名称、值

i	小整数 (Fixnum) 值
f	浮点数 (Float) 值
l	大整数 (Bignum) 值
"	字符串 (String) 字符串长度、文本
/	正则表达式 长度、模式
[数组 数组长度、值.....
{	哈希表 长度、名称、值.....
S	结构体 (Struct) 长度、名称、值.....
:	符号名 (symbol) 长度、名称
;	既有符号名 (symbol) id
I	类实例变量 对象、数、名称、值
@	参照id
c	类 名称
m	模块 名称

#dump 以下()内数据

p Marshal.dump([1, "2", {3=>4, 5=>6}])

#结果如下

#"\004\010[\010i\006"\0062{\007i\012i\013i\010i\011"

#以上结果的意义如下。(\\nnn 表示八进制数, 参照表13-2)

\004	主版本4	
\010	次版本8	
[\010	长度3 的数组<8 = 3 + 5>	
i\006	整数1<6 = 1 + 5>	...第1 要素
"\006	长度1 的字符串<6 = 1 + 5>	...第2 要素
2	长度2 的文本	
{\007	长度2 的哈希<7 = 2 + 5>	...第3 要素
i\012	整数5<10 = 5 + 5>	...键
i\013	整数6<11 = 6 + 5>	...值
i\010	整数3<8 = 3 + 5>	...键
i\011	整数4<9 = 4 + 5>	...值

#以上例子中, 整数是通过独特的变换方法来<表示>的

0	"0"
1~122	"n + 5"
-123~-1	"(n - 5) & 0xff" (位运算)
上述以外	"长 (1-4), 最大4 字节"

图 13-5 Marshal 数据的具体例子

Ruby 版本不同，Marshal 的变换格式也完全不同，所以对同一对象进行写入和读出时必须使用 Ruby 的同一版本。但是，同一主版本内，数据具有向下兼容性。

Marshal 的格式现已相当稳定，最近多年来没有变化。

13.1.6 不能保存的 3 类对象

Marshal 在实现上有限制。以下 3 类对象不能保存。

- 定义了特异方法的对象。
- 输入、输出或是套接字（Socket）等不能超越进程保存的对象。
- 在扩充库中定义，Ruby 不知道保存方法的对象。

Marshal 按类进行封送处理，所以，它不能处理含有“特异方法”的对象，因为这些方法的信息不属于任何类。

输入、输出及套接字等对象，只在特定进程单位内有效，不可能对它们进行合适的 Marshal。比如，进程一旦终止，文件或许被更改，或许被删除，不可能恢复到原来状态。

最后，由扩展方法定义的对象，由于构成 Marshal 的子程序不知道封送处理的方法，所以也不是封送处理的对象。

但是，即使不能够封送处理，若不是像输入输出那种从原理上不可能的情况，单纯是不知道封送处理方法的话，重新教一遍也就行了。Marshal 为用户定义对象提供了封送处理的方法。用户为每一个类定义一个用于封送处理的 `marshal_dump` 和用于复原的 `marshal_load` 方法就可以了。

如果想要 dump 的对象定义了 `marshal_dump` 方法，`Marshal.dump` 就使用该方法的结果进行 dump。`marshal_dump`

返回含有以后复原时必要信息的值，这个值被写入封送处理 **dump** 转储下来的数据中。

如果想要复原的对象定义了 **marshal_load** 方法，就用对应的 **marshal_dump** 生成的数据为参数，调出对象的 **marshal_load** 方法。**marshal_dump** 与 **marshal_load** 的使用方法示于图 13-6 中。

```
class Foo
  def initialize(a,b)
    @a = a
    @b = b
  end
  def marshal_dump
    [@a,@b]      #@a,@b 的值以数组形式dump
  end
  def marshal_load(data)
    @a = data[0] #用保存的值进行初始化
    @b = data[1]
  end
end
```

图 13-6 **marshal_dump** 和 **marshal_load** 的使用方法

13.1.7 制作面向对象数据库

使用 **Marshal** 保存对象，使对象具有了持久性。所以，**Marshal** 也可应用于面向对象数据库。

PStore 库（**persistent store**，对象持久性保存）是 **Marshal** 的用例之一。**Marshal** 虽然只是将数据变换为字节串，**PStore** 却利用了这一点，简单地实现了面向对象数据库。

因为对象的封送处理这一最麻烦的部分交给了 **Marshal** 模块，即使包含注释和空行，**PStore** 也只是一个仅有 400 行左右的小型库。**PStore** 是作为 **Marshal** 的用例来开发的，原型只有 100 多行，我只用了—个晚上就完成了。

PStore 有 3 个特征：使用 **Marshal**，可以原封不动地保存任意的 **Ruby** 对象；具有容易使用的接口；有事务处理（**transaction**）。

事务是数据库的处理单位。它是防止数据库变为不完整状态的一种机制。事务如果没有问题正常完成的话，其结果将正常写入数据库；如果由于某种原因（比如异常）中途失败，数据库的状态就是事务处理开始前的状态，没有变化。

PStore 也有缺点。它不适合一下子将数据全部读入内存的大规模数据库。但几百字节的小规模数据库，应该没问题。

13.1.8 试用PStore

按照顺序尝试一下使用 PStore 对象吧。

```
(a) 打开数据库
db = PStore.new(path)

(b) 开始事务处理
db.transaction{
  ...
}

(c) 对象的登录与取得
db["foo"] = object  #对象的设定
db["bar"]           #对象的取得

(d) 对象名的确认
db.roots            #root 名一览
db.root?("foo")     #root 名foo 存在时为真
```

图 13-7 PStore 的使用方法

打开数据库

首先像图 13-7a 那样打开数据库，得到一个 PStore 数据库对象。图中的 path 指定数据库文件的路径。

开始事务处理

如前所述，事务是数据库处理的单位。对数据库的处理是以事务为单位写入，如果处理途中发生了异常，数据库的状态将保持在事务处理

前的状态。事务处理通过在 **transaction** 方法中指定块来开始（参见图 13-7b）。

从块中退出，事务处理就结束了。

对象的登录和取得

PStore 对象可以作为一种哈希值来处理。像图 13-7c 那样，对象的登录和取得用 **[]** 方法。

但是，**PStore** 与哈希表还有区别，如果对象的名字没有登录就取得，就会出错。这样如果不知道一个名字是否在数据库里登录过，就有必要事先确认一下。

取得数据库中登录的全部对象名，用 **roots** 方法；检查某一名称的 **root**（根）是否登录了，用 **root?** 方法（参见图 13-7d）。

事务处理成功时对象的写入，是把带名称的对象作为根来进行的，这就是 **root** 这个词的来源。

事务处理的终止

transaction 方法中指定的块执行完之后，事务处理自动终止，数据库所引用的全部对象的状态都写入文件。

在事务处理的过程中，也有强制性终止的方法。**commit** 方法正常结束事务处理，所以，**commit** 后的执行位置就跳到 **transaction** 方法所指定块的末尾。**abort** 方法强制结束事务处理后，也在执行位置跳到 **transaction** 方法所指定的块的末尾，这一点上与 **commit** 相同，但数据库的变更不写入数据库，而是取消。

在 CGI 等多进程环境中使用 **PStore** 时，需要将数据文件加锁。**PStore** 在内部自动使用 **flock** 进行专用控制，这一点可以放心使用。但是，**PStore** 没有实现相同进程中的多个线程同时访问的专用控制。线程间的专用控制是程序员的责任。

数据库内对象的引用与更新，一定要在事务内进行。在事务处理中途取出的对象，在事务之外改变其状态虽然不出错，但这一变更不会反

映到数据库中（参见图 13-8）。

```
db_data = nil
db.transaction {
  db["foo"] = [1, 2, 3]
  db_data = db["foo"]    #取出
}
# 事务处理终止

# 可以访问事务处理内的数据
p db_data[0]             #输出[1]

# 虽然可以更新数据，但不反映到DB 里
p db_data[0] = 42
```

图 13-8 事务处理终止后访问数据库内对象的示例

简单说明一下事务处理的步骤。

1. 用 **flock** 将数据文件加锁。
2. 用 **Marshal** 从数据文件中读取数据。
3. 执行（事务处理）块。
4. 块的执行成功时，**Marshal** 将数据写入数据文件。
5. 块的执行失败时，什么也不做。

实际上，还应有备份文件的生成、错误处理等步骤，要更复杂一些，但基本上是这样的。

使用 **PStore** 的示例程序参见图 13-9。这个程序从标准输入接受名称，然后将各名称的出现次数录入数据库中。

```
require 'pstore'

# 打开数据库
db = PStore.new("/tmp/ncount")
#输入名称
STDOUT.print "input name:"
```

```

STDOUT.flush
name = gets.chomp

db.transaction do
  #名称不存在时的初始化
  db[name] ||= 0
  #名称的计数加1
  db[name] += 1
  #显示名称一览和计数
  db.roots.each do |n|
    printf "name: %s count: %d\n", n, db[n]
  end
end
end

```

图 13-9 使用 PStore 的程序示例

13.1.9 变换为文本的YAML

Marshal 的变换结果是二进制文件，内容不容易看懂。所以有些场合，即使效率低一点，也需要能够以更容易看懂的形式输出。

能够满足这种要求的是 YAML。YAML 是 YAML Ain't Markup Language（YAML 不是标记语言）的缩略语。这据说是开发 Perl Inline.pm 的 Brian Ingerson 研究出的数据序列化格式。

YAML 使用文本形式，不依赖于平台的体系结构，是一种对人而言易读易编辑的序列化格式。它提供了面向 Perl、Ruby、Python 以及 Java 等各种语言的 API，能够超越语言传递数据。

YAML 有以下几个特征：记述简洁；结果容易读懂；使用缩进的层次表现；数据表现是专用的，不必烦恼标签的名称问题。

YAML 可以活用在 Ruby on Rails 的配置文件等各种各样的领域。YAML 是在 Perl 中开始开发的，但正式的支持，Ruby 是第一个。Ruby 中 YAML 的基本使用方法如图 13-10 所示。

```

require 'yaml'
pack = obj.to_yaml # 将obj 变成YAML 后的字符串

# 从YAML 字符串复原为对象
unpack = YAML::load(pack)

```

图 13-10 YAML 的使用方法

加载 `yaml` 库以后，`Object` 类里就追加了 `to_yaml` 方法。调用这个方法，可以得到任意对象的 YAML 表现。图 13-5 的 `Marshal` 示例中使用的同样的值，我们给它变成 YAML 看看吧（参见图 13-11）。

```
require 'yaml'
print [1,"2",{3=>4,5=>6}].to_yaml

# 输出以下内容
---          # YAML 先头行
- 1          # "-"是指数组，先头要素是整数1
- "2"        # 第2 个要素是字符串2
- 5: 6       # 第3 个要素是哈希。哈希是"键:值"的排列
  3: 4       # 在数组内部，所以有缩进
```

图 13-11 YAML 数据的一个具体示例

YAML 用于配置文件时，很方便。而且，用 `to_yaml` 方法可以将任意对象变换为 YAML，可以干很多有趣的事情。使用 `yaml` 库，可以像 `Marshal` 一样简单地将各种对象通过 `dump` 转储为文本格式。而且，其表现比 `Marshal` 更易读，与 Ruby 以外的语言也可以进行交换。

13.1.10 用YAML制作数据库

与 `Marshal` 一样可以将对象与字符串进行相互变换，也就意味着可以实现使用 YAML 的面向对象数据库，即类似于 `PStore` 的东西。具体例子是 `YAML::Store`。`YAML::Store` 与 `PStore` 互换性非常高，只要把名字换一换，面向 `PStore` 的程序在 `YAML::Store` 中也能运行。

图13-9中的 `PStore` 示例程序，用 `YAML::Store` 重写以后，就成为图13-12那样。只要稍微改写一下，就能实现完全相同的动作。图 13-12中程序的变更点只有3点。一是 `require 'pstore'` 变成 `require 'yaml/store'`；二是访问 `PStore` 变成访问

YAML::Store；还有一个是数据文件的格式变了，所以文件名也变了。

```
require 'yaml/store'

db = YAML::Store.new("/tmp/ycount")
STDOUT.print "input name: "
STDOUT.flush
name = gets.chomp

db.transaction do
  db[name] ||= 0
  db[name] += 1
  db.roots.each do |n|
    printf "name: %s count:%d\n",n,db[n]
  end
end
end
```

图 13-12 使用 **YAML::Store** 的示例程序

像这样只要很少的变更，**PStore** 中能够运行的程序，在 **YAML::Store** 中几乎都能运行。事实上，把 **YAML::Store** 类做成 **PStore** 类的子类，只是将数据的整列部分更改为使用 **to_yaml** 方法。所以，提供 **YAML::Store** 功能的 **yaml/store.rb** 文件，算上注释和空行，也只有区区 30 行。

表面上的运行虽然完全相同，**PStore** 上运行的程序和 **YAML::Store** 中运行的程序还是有以下几点区别。

- **数据格式**

PStore 使用 **Marshal**，**YAML::Store** 使用 **YAML**。如果有可能有人去调查或操作数据文件的话，还是会发现 **YAML** 更易懂。

- **数据量**

像上面的示例程序这种小的数据，几乎没什么差别，但在封送处理大规模而又具有复杂结构的对象时，**Marshal** 比 **YAML** 紧凑得多。可以说，**Marshal** 牺牲了易读性而实现了良好性能。

- **执行速度**

性能优良不光是容量的问题。使用 Marshal 的 PStore 比 YAML::Store 速度快。在这一点上，也是数据量越大，两者的差异就越显著。

由于 PStore 与 YAML::Store 的性质有这些差别，所以有必要物尽其用地分开使用。一般遵照以下原则。

- 需要直接看数据内容时用 YAML::Store。
- 数据的委派目标不限于 Ruby 时用 YAML::Store。
- 有必要对用户定义的数据进行细微对应时用 PStore。因为 PStore 可以用 marshal_dump 进行定制。
- 数据量在一定程度以上，两种都不太合适的时候，可考虑导入专用数据库系统。

13.2 对象的保存

现在介绍一下对象持久化库 Madeleine。Madeleine 利用直接持久化¹对象的设计模式 Object Prevalence。

¹ 因为对象有参照关系，所以不能够单纯地变换为文本。

Madeleine (<http://madeleine.rubyforge.net/>) 是 Object Prevalence 在 Ruby 中的实现，应称为是前面介绍的 PStore²的发展形式，由 Anders Bengtsson 开发。

² PStore，是利用 Marshal 处理的库。Marshal 将对象进行文本化。

PStore 只是对象单纯地由 Marshal 输出而来，Madeleine 则与应用程序相协调，实现了高可靠性和高性能的持久化。

13.2.1 高速的Object Prevalence

所谓 **Prevalence**，是一种实现应用程序的持久化和进程间共享数据的设计模式。用 Java 的 **Object Prevalence** 库制作的 **Prevayler**

(<http://www.prevayler.org/wiki>)，与经由 **JDBC**³ 利用 **Oracle** 数据库的模式相比，速度不可思议地快了 9000 倍。

3 **JDBC** 是 Java 与数据库相连接的 API。顺便说一下，**JDBC** 不是缩写。

高性能的秘密，在于直接访问内存中的数据。**Object Prevalence** 中，将处理的数据保存在正在执行的应用程序的内存中，检索等操作不通过 **SQL** 而是直接进行。这样就节省了与数据库服务器的通信成本（处理时间），引用当然就会变得很高速。

但是，只有是同一进程，才能引用内存中的数据，进程一结束，数据马上就消失。从持久化的角度考虑，有必要解决这一问题。

Object Prevalence 用日志记录 (**journaling**)⁴ 和快照 (**snapshot**) 来解决这一问题。**Object Prevalence** 中，数据更新时不是直接更新对象，而是创建称为 **command** 的对象。它采用的是一种非常间接的方式，在用 **command** 更新对象的时候，内存中的对象更新的同时，所有的更新内容也会写到称为日志 (**journal log**) 的外部文件里。

4 所谓日志记录，是指这样一种记录方式：文件写入外部存储器时，在记录实际数据之前，先写入管理信息（元数据），以及元数据的变更记录。

这样长此下去的话，日志就会越来越大。所以，我们就定期地将现在数据的状态写入到称为快照的文件中去。有了快照，老的日志就不需要了，可以在适当的时机删除。

这里重要的是，有了最新的快照与最新的日志，就可以完全恢复现在对象的状态。程序启动时，按下面 3 个步骤可以恢复内存中的数据。即使有多个进程，只要写入日志中的信息是完整的，就可以共享对象的状态。

1. 如果不存在快照，就初始化应用程序数据。
2. 如果存在快照，就读入其中最新的一个。

3. 如果还存在日志，也将其读入，并用其中最新的一个更新应用程序数据。

即便是程序异常终止的情况，因为留有快照和日志，也还可能复原最新的数据。

基本原理就是这些，但 Java 版 Prevayler 中，另有别的 Java 虚拟机里保存有应用程序数据的副本（Replica），日志记录和快照的生成就交给了虚拟机。这样，就不必在每次取得快照的时候停止程序，从而实现了高性能。

Ruby 版的 Madeleine 中，还没有使用 Replica，为了取得快照，程序全体都要停止。所以，不能实现 Java 版那种特别高的性能。因为我们还没有在这种性能问题的应用上使用 Madeleine，所以还不太清楚它会有多大的影响。

13.2.2 Object Prevalence的问题点

Object Prevalence 通过使用日志记录和快照实现了对象的持久化和进程间共享，但它也不是没有缺点。Object Prevalence 将所有数据都保存到内存中，随着数据量的增大，内存的消耗也在增大。

关系数据库中，不引用的数据放在文件中，必要的内存量就不用那么多了。况且，最近内存的价格在下降，最大容量也在增加。普通的 PC 配以 GB 级的内存，也变得稀松平常了。也许内存的问题变得越来越不重要了。

另一个问题是使用 Object Prevalence 的程序，有着为了数据更新而具有的特殊结构。前面已经说明，更新持久化数据的时候，需要经由 command 对象。

这与我们平常习惯的直接更新对象的实例的做法大不相同，需要适应一下。后面会讲到对策。

13.2.3 使用Madeleine

Madeleine 作为设计模式 Object Prevalence 在 Ruby 中的实现，其使用方法示于图 13-13 中。这是一个简单的计数器程序。从键盘输入字符

串 **inc** , 计数器的值就增加, 输入字符串 **show** , 计数器的值就显示出来。

```
require 'madeleine'

class CountData
  attr_accessor :count
  def initialize
    @count = 0
  end
end

class CountInc
  def execute(data)
    data.count += 1
  end
end

class CountShow
  def execute(data)
    data.count
  end
end

# (A) 初始化Madeleine handle
m = SnapshotMadeleine.new("/tmp/data"){
  CountData.new
}

Thread.start {
  loop {
    sleep 120    #每120 秒取一次snapshot
    m.take_snapshot
  }
}

loop do
  print "inc or show? "
  STDOUT.flush
  line = gets
  break if line == nil
  case line
  when /^inc/
    printf "count -> %d\n",m.execute_command(CountInc.new)
  when /^show/
    printf "count:%d\n",m.execute_query(CountShow.new)
  end
end
```

图 13-13 Madeleine 的使用示例

为了利用 Madeleine，首先，像图 13-13（A）那样对系统进行初始化。SnapshotMadeleine.new 的参数里，指定快照和日志的存放路径。

可以省略的第 2 个参数指定将对象写入快照时所用的模块。缺省值是内置的 Marshal 模块。第 2 个参数若指定了 YAML⁵，可以保存为 YAML 形式。

⁵ YAML 是为了让人读起来更易懂而进行 Marshal 化的模块。

使用 zlib 可以将持久化数据压缩。首先，开头部分变为以下这样。

```
require 'madeleine'  
require 'madeleine/zmarshal'
```

在此基础上，SnapshotMadeleine.new 的第 2 个参数指定 Madeleine::ZMarshal.new(Marshal)。也可以使用 zlib 压缩的 YAML 文件，只要将参数 Marshal 变为 YAML 就行了。

SnapshotMadeleine.new 在快照存在的情况下，可以从快照复原应用程序数据。但是，程序在第一次启动时，因为不存在快照，就执行给定的块，把结果作为应用程序数据。图 13-13 的程序中，生成一个 CountData 类的对象。如果存在快照，就不执行块，而使用从快照和日志复原的 CountData 对象。不管是哪种情况，SnapshotMadeleine.new 的返回值都存放在 Madeleine 处理程序中。存放在 Madeleine 中的数据更新时，一定经由处理程序来创建 command，然后执行处理。具体访问方法如下。

```
m.execute_command(command)
```

`command` 被称为 `command` 对象，是执行数据访问的类。
`execute_command` 方法被调用后，按以下步骤处理：加锁，写日志以及调用 `command.execute`。

`command` 不管是哪个类的对象都无关紧要，但需要满足以下 4 个条件。

有 `execute` 方法

必须能够记述 `command.execute(data)`。`data` 是应用程序数据，由 `execute_command` 传递而来。

没有副作用

`command` 对象不能获取由 `execute` 的参数传递过来的数据以外的信息。全局变量、随机数、时刻的引用以及文件的输入输出也是禁止的。这是在按顺序调用保存在日志记录里的 `command` 对象的 `execute` 方法时，再现数据的 Object Prevalence 所必需满足的条件。

如果有引用全局变量这样的外部信息的 `command`，就不能从日志中复原数据。

可以持久化

`command` 对象由 `Marshal`（或者明确指定的持久化模块）写入日志。所以，`command` 对象本身必须能够持久化。

持久化的 `command` 对象写入日志记录，所以希望它本身不太大（尽可能不含对其他对象的引用）。

`command` 不含对 `data`（应用程序数据）的引用

`command` 被变成字符串记录到日志里，如果含有对应用程序数据的引用，日志文件就会膨胀，就根本谈不上性能了。前面提到过，`command` 不应含有太大的数据。

即使执行了 `command`，只要数据没有变更，还可以利用 `execute_query` 方法。`execute_query` 方法执行 `command` 时，

不往日志里写记录。因为日志里没留记录，执行的 **command** 不能更新应用程序数据。万一更新了数据，会成为很严重的问题。

再回到图 13-13。实际执行一下图 13-13 中的程序吧（参见图 13-14）。

```
$ ./ruby /tmp/m.rb
inc or show?inc
count -> 1
inc or show?show
count: 1
inc or show? inc
count -> 2
inc or show? ^D

$ ./ruby /tmp/m.rb
inc or show? show
count : 2
inc or show? inc
count -> 3
inc or show? inc
count -> 4
inc or show?^D
```

图 13-14 图 13-13 中程序的执行结果

中途按下 **Ctrl + D** 两个键，让程序结束。但是再次执行时，能够记住以前的状态（这里是 2）。

13.2.4 访问时刻信息

对 **command** 所作的限制中，禁止全局变量和文件的输入输出，就当做是没办法吧。随机数作为从外部传给 **command** 信息的一部分，也还能忍受。

但是，不能够访问时刻信息，对于编程则是一个相当大的限制。所以，我们准备了 **Madeleine::Clock::ClockedSystem** 模块。

为了使用这一模块，程序的开头写成这样：

```
require 'madeleine/clock'
```

在此之后，应用程序数据的类里，将这个模块包含进去，`clock` 方法就可以使用了。比如，现在时刻可以用`clock.time` 取得。

实际上，在取得时刻的时间点，时刻信息就写入了日志文件。但是，用户（程序员）没必要知道这一点。

`Madeleine::Clock::ClockedSystem` 的时刻记录程序用法示于图 13-15 中，每按下一次回车键，这个程序就记录一次时刻。

```
require 'madeleine'
require 'madeleine/clock'

class TimeRecord
  include Madeleine::Clock::ClockedSystem
  attr_accessor :last_time
  def initialize
    @last_time = nil
  end
  def record
    @last_time = clock.time
  end
end

class RecordCommand
  def execute(data)
    data.record
  end
end

m = SnapshotMadeleine.new("/tmp/record"){
  TimeRecord.new
}

# (A) 内部时钟的初始化
Madeleine::Clock::TimeActor.launch(m)
loop do
  last = m.system.last_time
  puts last if last
  print "press enter to record time "
  STDOUT.flush
  line = gets
  break if line == nil
  m.execute_command(RecordCommand.new)
end
```

图 13-15 使用 `Madeleine::Clock::ClockedSystem` 的时钟程序

当然，即使中断进程，只要留下日志记录和快照数据，下次启动时还能记得最后记录的时刻。

利用 `madeleine/clock` 库时应当注意的是，像图 13-15 (A) 所示那样，别忘了调用 `Madeleine::Clock::TimeActor.launch()` 方法。调用时，作为参数，传递 `Madeleine` 对象的处理权。

这一步骤对于内部时钟的管理也是必要的。如果忘了调用这一步，时钟就不能初始化，会给出一个怪异的时间（具体讲，总是 1970 年 1 月 1 日上午 9 时）。

13.2.5 让 `Madeleine` 更容易使用

正如到目前为止所见到的那样，`Madeleine` 既保持简洁性与高性能，又能让对象持久化，但它也有缺点。

要说最大的缺点，还是在每次更新应用程序数据时，都必须要生成 `command` 对象，调用 `execute_command`。

通常在 `Ruby` 中，对象的操作单纯是通过调用方法来实现的。相比之下，`Madeleine` 的这一缺点会让人很不耐烦。

这里将要介绍的是附属于 `Madeleine` 的 `madeleine/automatic` 库。用这个库，应用程序数据的更新可以通过普通的方法调用来实现。

图 13-16 是图 13-13 中的程序经改良而来的。使用 `madeleine/automatic`，对应用程序数据的操作变为普通的方法调用。`execute_command` 的调用没有了，可以直接调用用 `Madeleine` 对象的 `system` 方法取得的应用程序数据（`CountData` 类的对象）的方法。数据操作部分变得更直接更易懂。

```
require 'madeleine'
require 'madeleine/automatic'
```

```

class CountData
  include Madeleine::Automatic::Interceptor
  def initialize
    @count = 0
  end
  def show
    @count
  end
  automatic_read_only :show
  def inc
    @count += 1
  end
end

m = AutomaticSnapshotMadeleine.new("/tmp/data"){
  CountData.new
}

Thread.start {
  loop {
    sleep 120 # 每120 秒取一次快照
    m.take_snapshot
  }
}

loop do
  print "inc or show?"
  STDOUT.flush
  line = gets
  break if line == nil
  case line
  when /^inc/
    printf "count -> %d\n", m.system.inc
  when /^show/
    printf "count: %d\n", m.system.show
  end
end

```

图13-16 用madeleine/automatic 库改良图13-13后的程序

另一方面，表示应用程序数据的 `CountData` 类，将 `Madeleine::Automatic::Inter-ceptor` 模块包含进来了。这一模块实现了“魔法”，将方法调用在内部置换为创建 `command` 。

我们仅仅为了取得现在状态的 `show` 方法，而不更新数据，所以没有必要在日志里留下记录。这里追加了 `automatic_read_only` 设定，可以削减多余的日志。使用 `madeleine/automatic`，可以让 **Madeleine** 的带持久化功能的应用程序开发变得更简单。

13.2.6 **Madeleine**的实用例Instiki

虽然 **Madeleine** 具有各种各样的优点，却没怎么得到广泛应用。除了知道的人不多之外，还因为数据全保存在内存里，就必须十分留意数据的大小，这也成为妨碍它广泛使用的因素。

实际上，利用**Madeleine**的应用程序的代表是wiki的实现之一**Instiki**。**Instiki**是因开发Ruby on Rails而出名的David Heinemeier Hansson在发布Ruby on Rails之前所开发的程序。

Instiki 在利用 **Madeleine** 方面也是一个很有意义的应用程序。另外，它也反映了 MVC 构成的目录结构，安装非常简单，体现了后来的 Ruby on Rails 的一些特征，是个很有趣的软件。

但是，**Madeleine** 有一个很大的缺点，那就是没有考虑多个进程同时更新数据的情况。

比如并行执行图 13-13 中介绍的计数器程序，计数就会出现错位（参见图 13-17）。在图 13-17 中，终端 A 中图 13-13 的程序正在执行，又从终端 B 启动这一程序。看起来终端 A 的显示内容不受终端 B 的影响，但结束终端 A 的程序后再度启动，记录的数值就会从 2 增加到 4。因此，有必要采取对策，让使用 **Madeleine** 的程序服务器化。

```
# 终端A
$ ./ruby /tmp/m.rb
inc or show? inc
count -> 1
inc or show? show
count: 1

# 终端B
$ ./ruby /tmp/m.rb
inc or show? inc
count -> 2
inc or show? inc
```

```
count -> 3
inc or show? ^D

# 终端A (续)
inc or show? inc
count -> 2
inc or show? ^D

$ ./ruby /tmp/m.rb
inc or show? show
count: 4
```

图 13-17 计数器程序同时执行的示例

13.3 关于 XML 的考察

以前，我曾经写过对 XML 持批判态度的博客。所以，很多时候我被视为 XML 反对派。的确，很多人不管合适不合适，到处都用 XML，这让我很困惑。但这顶多是个使用场合的问题，我无意否定 XML 技术本身。

这一节考察一下 XML 的性质，分析其长处短处，知道在哪些地方该用，哪些地方不该用。

13.3.1 XML的祖先是SGML

首先回顾一下 XML 的技术史。XML 的祖先是 SGML（Standard Generalized Markup Language）。SGML 是将文档电子化的一种格式。它由 3 部分组成：表示数据本身的 Instance，表示数据结构格式的 DTD（Document Type Definition），以及 SGML 声明。

SGML 最初由美国 IBM 公司的 Charles Goldfarb 所开发，1986 年通过国际标准化组织（ISO）的认证，其规范文件为 ISO8879。在那之后，它被广泛应用于产品手册等各种文档的电子化上。

WWW 诞生以后，网页的表现形式采用了 SGML。HTML（Hyper Text Markup Language）是 SGML 适用于特定的 DTD 的产物，也可以说是 SGML 的实例，或者说是 SGML 的子集。

由于 SGML 太复杂，处理成本太高，为了表现网页，我们将 SGML 特化为 HTML，随之而诞生的是 XML（Extensible Markup Language）。

XML 不像 HTML 那样是为了特定目的的标记语言，它一开始就是为了通用目的而设计的。另外，为了让 XML 在没有 DTD 来定义语法或提供 schema 信息的情况下，也能够解析，人们对其语法进行了简化。XML 的规范于 1995 年左右在 W3C 协会（World Wide Web Consortium）上开始讨论，1998 年发布 XML 1.0。2004 年公布的 XML 1.1 版是最新的规范。

13.3.2 XML 是树结构的数据表现

现在再来看看 XML 的概要吧。XML 数据的形式如图 13-18 所示。这是假想地址信息的 XML 数据。XML 基本上是纯文本，以类似于 HTML 的标签将文本括起来的形式来附加信息。另外，以标签嵌套的方式实现树结构。

```
<addressbook>
  <address>
    <name initial = "M">松本行弘</name>
    <zipcode>699-0201</zipcode>
    <address>松江市玉汤町玉造</address>
    <tel>0852-62-XXXX</tel>
  </address>
  <address>
    <name initial = "N">网络应用通信研究所
  </name>
    <zipcode>690-0826</zipcode>
    <address>松江市学园南2-12-5</address>
    <tel>0852-28-XXXX</tel>
  </address>
  <address>
    <name initial = "R">乐天</name>
    <zipcode>140-0002</zipcode>
    <address>品川区东品川4-12-3</address>
    <tel>03-0387-XXXX</tel>
  </address>
</addressbook>
```

图 13-18 XML 数据的示例

如上所述，XML是继承了SGML的通用标记语言。它与SGML最大的区别是其基本语法固定，不依赖于DTD那样的外部信息也能解析。XML 中规定，像<data> 对应</data> 那样，标签必须用同名的结束标签来结束；没有结束标签的（空要素标签），像
 那样，末尾必须带斜杠。

同样是继承自 SGML，HTML 就没有这种规则，比如<hr> 就没有结束标签。因为不知道标签在哪里结束，如果没有各标签的概要信息，就不能解析。

像这样即使没有标签的概要信息也能解析的语法称为良构的（well-formed），这是 XML 的一大特征。

13.3.3 优点在于纯文本

XML 有很多优点，其中最大的优点在于 XML 基本上是纯文本的。想一想所表示的数据，比如说用*.doc 这种 Word 格式的数据，随着时光流逝，当处理这种数据的软件再也找不到的时候，信息就丢失了。XML 从 SGML 继承的好处就是，数据全部以纯文本表示，表示结构的信息附加在标签里。

第 2 个优点是不易发生关于字符编码的问题。Tim Bray 是开发 XML 的初期成员之一，有时被称为“XML 之父”。我曾经有过一次与他亲自对话的机会，当被问到“您认为 XML 的最大优点是什么？”的时候，他马上回答“没有字符编码的问题。”

XML 规定，在没有明确指定的情况下，字符编码均使用 Unicode。在没有这一规定，编码信息不一定存在的环境中，比如初期的 HTML 中，文本数据的编码方式只能靠推测，万一推测错了，就会带来严重的乱码问题。现在，HTML 中指定编码方式也已经成为理所当然的事情，遇到乱码的机会变少了。XML 把这种看似理所当然，而实际并非理所当然的东西，清清楚楚进行了定义，是很了不起的。

第 3 个优点是，得益于良构的性质，它在没有数据结构的情况下也能解析 XML 数据。这样就可以不考虑目的，而用共同的工具来处理 XML 数据。当然，根据需要，使用数据结构信息（称为 schema），也可以验证 XML 数据是否具有合乎目的的结构。

第 4 个优点在于，XML 与其解析工具不依赖于特定的语言，比如 Java 生成的 XML 数据在 Ruby 中解析也很简单。解析 XML 的 API，像 DOM 和 SAX（Simple API for XML），都超越语言提供了几乎共通的内容，所以不同的语言也可以进行同样的操作。因为这个性质，不同平台之间的信息交换性很好，也就是说，XML 的互用性很高。

因为 XML 不依赖于语言，且是纯文本的，所以说它对于平台的独立性非常高。这意味着即使处理数据的软件更新了，数据仍可以继续使用。

最后一个优点是，从 SGML 继承而来的纯文本+标签的语法，能让人马上明白数据具有什么样的结构，也即对人而言是一种容易理解的结构。还有，开始和结束标签具有相同的名字，如果因弄错而失去了正确的嵌套结构，就可以马上检查出来，这样能迅速发现和修正编辑的错误。

总结一下，XML 作为各种数据交换格式的框架，具有优良的性质。换言之，作为格式的格式，也就是元格式，它是很优秀的。

XML 是不是最好的元格式，还有异议，也有人认为 Lisp 采用的 S 式更优秀。但是，S 式作为信息交换的格式，没有被规范化，这无疑是它相对于 XML 的一个劣势。

13.3.4 缺点在于冗长

有这么多优点的 XML，也并非完美无瑕，批判它的人也不少。

XML 最为人所诟病的是效率低下。XML 是以纯文本形式表现的，标签信息反复出现，显得很冗长。与表示相同信息的二进制数据相比，XML 数据的容量要大得多。而且，XML 与其他文本表现方式相比（比如 YAML、JSON 等，后面会讲到），也要显得冗长。

效率低下不光是体现在数据大小上，解析 XML 时效率也不怎么高。与二进制文件相比，XML 文件的解析因为含有大量的字符串处理，而容易变得很慢。为了解决这一问题，有人提议过用 XML 的二进制来表现，但那样的话又会失去它的一个优点，即一看就懂的易读性。万一有什么差错的时候，也无法安心地用文本编辑器来编辑了，真让人左右为难。

另外，XML 虽然号称自己能够表现数据结构，对人而言易读，但是实际上现在已经认识到，复杂到一定程度以上的 XML 数据，特别是有一定结构的数据，靠人去读是很不现实的。一般不是靠人去读写 XML，而是用某种工具去操作它。

说起冗长的标签所带来的容易检出错误这一好处，实际上，人们在编辑 XML 时一般都使用工具来操作，所谓易读、易编辑这些优点，也只是在出了什么问题，需要紧急处理，靠人为操作时才能体现出来。

说得更深一点，作为文本的标记语言而诞生的 XML，用其表现有一定结构的数据到底好不好还是一个疑问。如果只是用于表现构造数据，比 XML 更有效率的格式还有好几种呢。而且，XML 原则上只能表现树结构的数据，这也是很让人遗憾的。

总结一下，XML 作为出于通用目的的数据格式，效率很低。很多所谓的优点，如果场合不对，也没多大意义。有一个说法叫适材适所，XML 也有必要分情况适当使用。

13.3.5 不适合重视效率的处理

知道了这些优点和缺点，也就明确了 XML 的“用武之地”。首先，XML 不适合的，还是重视效率的地方。XML 的冗长性和解析效率的低下性，对于重视通信量和速度的情况都不合适。这些情况下，使用专用的协议或者效率更高的格式，应该更好。

另外，对于像配置文件那样的靠人直接编辑的数据，也不推荐使用 XML。配置文件里，需要用到 XML 的树结构数据的地方很少，随着要素数的增加，它很快就变得很难读，不适合用 XML。如果用于这种目的，后面将要讲到的 YAML 和 JSON 那种更简单的格式才合适。

以上光说了缺点，那么 XML 在哪些场合才真正合适呢？

看看上述那些缺点的反面：

- 人一般不直接接触；
- 复杂性不成为问题；

- 效率不成为问题;
- 跨平台。

以上这些场合是适合使用 XML 的场所。复杂性不应成为问题，是指将 XML 作为本来的标记语言使用的时候，标签的密度不大，所以不易成为“读不懂的 XML”。

对于作为数据进行处理的情况，如果是读不懂的 XML，人眼一看就能知道读不懂。即使这样也要用的时候，是不是对于跨越系统处理数据以及数据的跨平台性有要求哇？比如说，多个系统共享数据（而且不是频繁访问）的时候。

13.3.6 适合于信息交换的格式

想想这些，会让人觉得 XML 其实也没有所说的那么万能。即使这样，仍然有几种适合使用 XML 的情况，其中之一就是以 XML 为基础的数据格式。利用 XML 的元格式性质，定义了多个以 XML 为基础的格式。基于这种信息交换用的格式，参与者之间的交换协议是最重要的。

以下列出了几个以 XML 为基础的格式的例子。

- RSS。Web 网站更新信息;
- Atom。RSS 的代替;
- ebXML。电子商务数据交换;
- SVG。向量—图像表示;
- SMIL。多媒体及内容控制。

以上这些都具有 XML 的性质，可以用 XML 处理工具简单地解析。制作数据格式时，最麻烦的就是制作处理这种格式的软件，所以，XML 与 XML 处理库（及工具）的存在是很宝贵的。

另外一点是 XML 数据库这样的构造。XML 数据库中，问题不在于数据是不是实际上以纯文本的 XML 来表现，而在于 XML 能够表现的树结构能够自由自在地操作。也就是说，不是带标签的纯文本，而是由带属性、带内容的节点所构成的树结构本身才是重要的。这样，关系数据库的表只能间接表现数据，如果是树结构，可以直观操作直接表现的数据。

XML 数据库中，数据无非是保存在数据库中（估计是用二进制表示），表现的冗长性和效率的低下性不成为问题。倒是后面要讲到的 DOM 那样的 API 中重视树结构的操作。

13.3.7 XML的解析

XML 的解析方法有好几种，这里介绍 DOM、SAX 和 XPath，Ruby 中提供这些功能的库 REXML 也一并予以介绍。

DOM

DOM 是文档对象模型的缩写，是对读取了 XML 数据的树结构进行操作的库。DOM 提供的操作由 W3C 作为标准进行了定义。

SAX

SAX 是 Simple API for XML 的缩写。与将数据全部读入内存的 DOM 不同，通常，SAX 以数据流的形式读入 XML，以事件驱动进行处理。SAX 中，没必要将数据全部读入，这样往往处理效率更高，所以，适合于将 XML 变换为其他形式的处理，反过来说，不适合于对树结构进行随机访问等用途。

XPath

XPath 是用于指定 XML 树的一部分的书写格式。使用 XPath，可以用节点名（标签名）、属性名或属性值等来选择特定的节点（群）。

13.3.8 XML处理库REXML

REXML 是 Ruby 标准附属的 XML 处理库。REXML 是具有 DOM、SAX、SAX2 以及 XML Pull Parser 等多种功能的库，由住在德国的

Sean Russel 先生维护。

REXML 全部用 Ruby 实现，所以速度表现上不怎么优秀。在特别重视效率的情况下，也许有必要用 libxml 等别的 XML 处理库。

现在来看一看实际使用 REXML 的程序吧。首先从使用 DOM 的程序开始。图 13-19 的程序从图 13-18 所示的地址信息的 XML 数据中，提取出姓名和住所并显示出来。为了使用 DOM 读取 XML 数据，需要将 `rexml/document` 库加载进来，接着生成供 DOM 操作的 `REXML::Document` 对象，然后可以自由访问树结构。`elements` 方法返回该节点下配置的节点。通过明确指定名称，也可以只选择特定的节点。图 13-19 中程序的执行结果示于图 13-20。

```
require 'rexml/document'

doc = REXML::Document.new(File.open("address.xml"))

doc.elements.each("/addressbook/address") do |e|
  print e.elements["name"].text, " - ",
        e.elements["address"].text, "\n"
end
```

图 13-19 使用 DOM 的程序

```
松本行弘 - 松江市玉汤町玉造
网络应用研究所 - 松江市学园南2-12-5
乐天 - 品川区东品川 4-12-3
```

图 13-20 DOM 示例程序的执行结果

DOM 也可用于构建树结构（参见图 13-21）。可以知道，这个程序执行以后，会新增一个 **Ruby association** 节点。

```
require 'rexml/document'

doc = REXML::Document.new(File.read("address.xml"))

address = doc.root.add_element("address")
e = address.add_element("name", {'initial' => 'R'})
e.text = "Ruby association"
```

```

e = address.add_element("zipcode")
e.text = "690-0003"
e = address.add_element("address")
e.text = "松江市朝日町478-18"

doc.root.elements.each("/addressbook/address") do |e|
  puts e.elements["name"].text
end

```

图 13-21 由 DOM 生成的树结构

DOM 可以从 XML 数据实现树结构，并进行操作。另一方面，SAX 一边读取 XML 数据，一边根据所遇到的标签或内容，按事件驱动来处理 XML。对于所发生的事件，调用相应 **Listener** 对象的方法。在 **Listener** 类里定义想要处理的事件所对应的方法，在每次该事件发生时就调用那个方法。

SAX 的基本使用方法示于图 13-22 中。SAX 产生的事件示于表 13-3 中。

```

require 'rexml/document'
require 'rexml/streamlistener'
class MyListener
  include REXML::StreamListener
  def tag_start(name,attrs)
    print "tag:",name
    if attrs.size > 0
      print ", ", attrs.inspect
    end
    print "\n"
  end
end
#
#)
def text(content)
  unless content.strip == ""
    print "text: ", content, "\n"
  end
end
end
filename = ARGV[0]
REXML::Document.parse_stream(File.open(filename),MyListener.new)
↑以Listener 类为参数，调用parse_stream

```

←Listener 类声明

←include StreamListener

←定义事件对应的方法

每次事件发生时调用的参数

随事件而不同，比如tag_start

标签开始时，参数是标签名

和属性（哈希）

←本文（纯文本）对应的方法，参数是

本文内容（字符串）

图 13-22 SAX 的使用方法

表13-3 SAX的事件

事 件	方 法 名	参 数
notation声明	notationdecl	content
标签开始	tag_start	name, attrs
标签结束	tag_end	name
文本	text	text
处理命令	linstruction	name, instruction
注释	comment	comment
文本型声明	doctype	name, pub_sys, long_name,uri
文本型声明结束	doctype_end	无
属性list声明	attlistdecl	element_name, attributes, raw_content
元素声明	elementdecl	content
实体声明	entitydecl	content
notation声明	notationdecl	content
实体	entity	content
CDATA	cdata	content
XML声明	xmldecl	version, encoding, standalone

图 13-22 中的程序，在与开始标签和文本相对应的事件中输出有关的信息。为了让结果更容易看，一不表示空属性，二不表示只有空白的text。对于图 13-18 中的 XML 数据的执行结果示于图 13-23 中。

```
tag:addressbook
tag:entry
tag:name, {"initial"=>"M"}
text:松本行弘
tag:zipcode
text:699-0201
tag:address
text:松江市玉汤町玉造
tag:tel
text:0852-62-XXXX
tag:entry
tag:name, {"initial" => "N"}
text:网络应用通信研究所
```

```
tag:zipcode
text:690-0826
tag:address
text:松江市学园南2-12-5
tag:tel
text:0852-28-XXXX
tag:entry
tag:name, {"initial" => "R"}
text:乐天
tag:zipcode
text:140-0002
tag:address
text:品川区东品川4-12-3
tag:tel
text:03-0387-XXXX
```

图 13-23 SAX 的示例程序的执行结果

作为使用 SAX 的一个有实用性的例子，试着写一个程序，检查所给的（像是）XML 的数据是否是良构的。图 13-24 中的程序用了 SAX，但 `Listener` 类中没有定义任何方法。这样，就不处理任何事件，仅仅利用了 SAX 的 `stream parser` 来检查能否正常读进来。

```
require 'rexml/document'
require 'rexml/streamlistener'
class WellFormedChecker
  include REXML::StreamListener
end
filename = ARGV[0]
begin
  REXML::Document.parse_stream(File.open(filename),
  WellFormedChecker.new)
  printf "%s is well-formed\n", filename
rescue REXML::ParseException => e
  printf "%s:%s", filename, e
end
```

图 13-24 XML 良构性检查

图 13-24 中程序的执行结果示于图 13-25 中。图 13-25a 是当参数指定的 XML 为良构时的输出。图 13-25b 是非良构时（损坏了）的输出。

(a) well-formed 的情况

```
% xmlcheck.rb address.xml
address.xml is well-formed
```

(b) 损坏时的情况

```
% xmlcheck.rb broken.xml
broken.xml: Missing end tag for 'address'(got "addressbook")
Line:19
Position: 564
Last 80 unconsumed characters:
```

图 13-25 XML 良构性检查程序的执行结果

最后，REXML 中 XPath 的使用方法示于图 13-26 中。XPath 在从 XML 树中抽取满足条件的节点时非常方便。

```
require 'rexml/document'
doc = REXML::Document.new(File.read("address.xml")) ←取得最初的
address 节点
puts REXML::XPath.first(doc, "//address") ←输出:<address>松江市玉汤
町玉造</address>
puts REXML::XPath.match(doc, "//name") ←取得全部name 节点, 返回数
组
# 输出:
# <name initial = 'M'>松本行弘</name>
# <name initial = 'N'>网络应用通信研究所</name>
# <name initial = 'R'>乐天</name>
puts REXML::XPath.match(doc, "//address/name") ←取得address 节点的子
节点name 节点, 返
# 输出:                                回数组, 这次的XML 与
上面一样
# <name initial = 'M'>松本行弘</name>
# <name initial = 'N'>网络应用通信研究所</name>
# <name initial = 'R'>乐天</name>
REXML::XPath.each(doc, "//address[name/@initial='R']") do |e| ←对于
节点name 的属性initial 是
puts e                                R 的
全部address 节点进行循环,
# 输出:                                这次
的XML 中只有一个这样的节点
# <address>
# <name initial = 'R'>乐天</name>
# <zipcode>140-0002</zipcode>
# <address>品川区东品川4-12-3</address>
# <tel>03-0387-XXXX</tel>
# </address>
```

```
end
```

图 13-26 XPath 的使用方法

除此之外，REXML 还有使用 RelaxNG 的验证（validation）等为数众多的功能，这些就不介绍了，以后有机会再说吧。

13.3.9 XML 的代替

光说“XML 并非万能”却不介绍其替代品，也许是不负责任的。这里介绍一下在各种不同的情况下，比 XML 更合适的技术。因为这次主要是介绍 XML 的，所以对于其他技术只介绍名称和概要。有兴趣的请自行查阅一下。关于 JSON 和 YAML，本书第 5 章有讲解。

JSON (JavaScript Object Notation)

JSON 是就把 JavaScript 的对象记法作为数据表现格式来使用。JSON 数据的例子示于图 13-27。

```
[
  {
    "initial": "M",
    "name": "松本行弘",
    "zipcode": "699-0201",
    "address": "松江市玉汤町玉造",
    "tel": "0852-62-XXXX"
  },
  {
    "initial": "N",
    "name": "网络应用通信研究所",
    "zipcode": "690-0826",
    "address": "松江市学园南2-12-5",
    "tel": "0852-28-XXXX"
  },
  {
    "initial": "R",
    "name": "乐天",
    "zipcode": "140-0002",
    "address": "品川区东品川4-12-3",
    "tel": "03-0387-XXXX"
  }
]
```

图 13-27 JSON 数据的例子

图 13-27 中的数据相当于图 13-18 中的 XML 数据，但与 XML 不同，没有明确的标签，在表现上稍微快一点。另外，JSON 不是标记语言，不适合于表现附加了信息的文本那样的半结构数据。

将 JSON 数据原封不动地作为 JavaScript 去执行，就可以得到数据表现所对应的对象。但是 JSON 数据从外部读取的情况较多，实际上作为 JavaScript 直接执行容易引起安全上的问题，即使效率稍微低一点，也应当使用解析 JSON 的库。

正如其名，JSON 是 JavaScript 的对象表现，作为数据表现语言，其语法及意义都定义得很清晰，所以 Ruby 及其他语言都支持它。

YAML (YAML ain't Markup Language)

前面介绍过的 YAML，是一种数据格式，有着一个不知其所云的名字，即所谓“YAML 不是标记语言”。YAML 是作为 XML 的对立面而诞生的，具有以下特征。

完全放弃标记性记述，专注于数据表现；以缩进为基础表现数据结构；不要标签；可以对应各种语言。结果，在用作数据表现及配置文件时，具有易读和不易变复杂等优点。实际上，YAML 在 Ruby on Rails 中广泛用于配置文件。

另一方面，YAML 到底还是数据表现语言，没有相当于 schema 的东西，不适合于带结构的文本表现及元数据格式。YAML 数据的例子示于图 13-28。

```
- initial: M
  name: 松本行弘
  zipcode: 699-0201
  address: 松江市玉汤町玉造
  tel: 0852-62-XXXX
- initial: N
  name: 网络应用通信研究所
  zipcode: 690-0826
  address: 松江市学园南2-12-5
```

```
tel: 0852-28-XXXX
- initial: R
name: 乐天
zipcode: 140-0002
address: 品川区东品川4-12-3
tel: 03-0387-XXXX .
```

图 13-28 YAML 数据的例子

活用记号和缩进的 YAML，比 JSON 更简洁。考虑一下是否采用易读、易编辑的数据格式 YAML 吧，应该不错的。但正如其名，YAML 不是标记语言，需要使用标记语言的时候还是 XML 合适。

Binary XML

XML 效率低的原因在于，为了在万一遇到什么问题的时候让人容易读，采用了冗长的纯文本。这样做虽然有这样做的好处，但效率低下的问题怎么都不好解决。

这里，与通常的 XML 有等价意义，但效率更高，采用二进制表现的是 Binary XML。但现状是还没有 Binary XML 的标准规格，只有多种实现版本。

Protocol Buffer

Protocol Buffer（协议缓冲）是 2008 年美国谷歌（Google）公司作为公开源代码发布的数据表现形式（及相关工具软件）。Protocol Buffer 是很久以来就在谷歌内部使用的进程间通信技术。在谷歌数据中心的大量计算机中运行的进程之间，进行着大量的信息交换处理。这些进程有的以 Java 记述，有的以 C++ 记述，有的则以 Python 记述。为了高效通信，谷歌采用二进制表现它，时间和空间效率都有提高，这是一种灵活的表现。

Protocol Buffer 中，使用了一种“数据描述语言”来定义数据结构，然后从这个定义生成一个库，将原始数据变换为二进制表现（序列化）。

与此相类似的技术，还有美国 Facebook 公司开发的，后来作为开放源代码的 Thrift。与仅仅做序列化的 Protocol Buffer 不同，Thrift 连远程过程调用（remote procedure call）都包括了。

* * *

本节介绍了 XML 的优点与缺点以及其适用的领域。另外还介绍了在 XML 不适合使用的领域中可以替代 XML 的技术。

持久数据的重要性

世界上充满了各种格式的数据。表示图像的.jpg，表示字处理文档格式的.doc，表示用于演讲的幻灯片的.ppt 等。每出现一种新的软件，就增加了一种新的数据格式。难以想象世上到底有多少种数据格式。

但是，打开某些数据格式会成为十分棘手的问题。有一天，由于某种原因我需要取出自己在学生时代曾经使用的非常古老的数据。那是在现在已经不再使用的计算机中运行过的，现在已经不再使用的字处理软件的数据。数据就在眼前，但软件不存在了，就无法读取数据，也无法加工处理。当时，也没有别的办法，我只能根据侥幸残留下来的打印纸，一边看一边再输入一次，总算对付过去了。从那以后，我对于未知的数据格式，总有一种挥之不去的不信任感。将来可能会用到的数据，一定要做一个文本数据。

历经岁月流逝仍然安全的数据格式，应该是文本文件吧。如果是文本文件，即便过了 10 年、20 年、30 年，依然能读。实际上，很久以前，我在学生时代所作的毕业论文，到了现在还能读出。即使像 ASCII → EUC-JP → UTF-8 这样，字符编码方式变迁了，也不至于完全读不出数据。同样，采用纯文本的电子邮件，这 30 年来，基本构造也没有变化。能有这样的安全稳定性真是了不起。

本章虽然对于 XML 有若干批判性的记述，但至少 XML 能够作为纯文本读入，经得起长时间保存这一点，还是应该给予正面评价的。

话虽如此，即便数据经得起保存，但保存数据的媒介却又成了问题。比如说现在已经找不到软盘驱动器了。

不能确定现在主流的硬盘和闪存究竟能用到什么时候。我学生时代的数据，虽然几乎都拿出来保存在磁带里了，但已经没办法读出来了。包括我最初开发的编程语言（不是 Ruby）的源代码，等于已经失去了。

这样看来，纸对于人类文明来说，真是一个伟大的发明。如果不是有了像纸和刻了文字的石头等经久不烂而且可以读出的媒介，将来人类文明说不定会遇到失去重要信息的危险。

第 14 章 函数式编程

14.1 新范型——函数式编程

函数式编程是全部使用函数来编写程序代码的编程方法。这是可以与一般的结构化编程（C 或者 Java）及面向对象编程相提并论的编程方法。

关于函数式编程这种叫法，一般认为起源于 FORTRAN 的开发者 John Backus 于 1977 年在图灵奖授奖仪式上所做的讲演，他介绍的编程语言 FP 是首先使用这个名字的。

以函数为中心的函数式编程具有如下的特征：

- 函数本身也作为数据来处理（第一级函数）；
- 以函数为参数的高阶函数；
- 参数相同即可保证结果相同的引用透明性；
- 为实现引用透明性，禁止产生副作用的处理。

函数式编程的最大优点在于，程序可以按照数学的形式以及声明的形式来编写。

谁也不会否认计算机的基础是数学。不管是算法，还是计算机科学的基础部分，都是与数学密不可分的。函数是数学领域已经使用了几百年的概念，非常适合于用数学的形式来表述。

举个例子吧。大家都知道阶乘的概念，数学上用 $n!$ 来表示阶乘，它是 1 到 n 的所有整数的乘积。根据这个定义，3 的阶乘等于 $1 \times 2 \times 3$ ，结果是 6。稍微改进一下，用数学的形式来写的话，就变成下面的式子：

$n! = 1$	n 等于 1 的时候
$n! = n \times (n-1)!$	$n > 1$ 的时候

这个阶乘的定义是利用归纳法，通过阶乘本身来定义阶乘。

(a) 阶乘计算的结构化程序

```
def fact(n)
  fct = 1
  while n > 1
    fct = fct * n
    n = n - 1
  end
  fct
end
```

(b) 阶乘计算的函数型程序

```
def fact(n)
  if n == 1
    1
  else
    n * fact(n-1)
  end
end
```

图 14-1 阶乘计算的程序

图 14-1a 是用结构化编程方式编写的阶乘计算的程序。在一次次减小 n 的同时，把 n 和变量 fct 的乘积赋给变量 fct 。有编程经验的人，差不多可以想象出局部变量 n 和 fct 的值在计算过程中变化的样子。

另一方面，图 14-1b 是用函数式编程方式来编写的阶乘计算的程序。仔细看一下就会明白，这种程序的写法与上边阶乘的归纳法定义几乎完全一样。像这样，函数式编程可以把算法写得非常接近于数学的表达形式。

但是，并不是所有的软件都是以数学算法为中心的，其他时候函数式编程也能有令人欣喜的表现吗？

信奉函数式编程的人相信，不管在任何场合，函数式编程都是有益的，而这种信念确实有其正确性。

让我们再回头来看看图 14-1 的程序。采用结构化编程方式编写的程序 (a) 是在改变变量值的同时进行计算的。因此，需要一直注意着，这个变量的值现在是什么，并据此来预想计算过程。

另一方面，采用函数式编程方式 (b) 并不改变变量的值。实际上它只是把阶乘的定义“整数 n 和它的阶乘之间是这样一种关系”换了个描述方式而已。这种编程方式中并不包含状态或者动作等信息，仅仅是对想要做什么加以描述，这样不容易出错。

这种不是描述动作，而是描述性质的编程方式称为声明式编程。声明式描述是函数式编程的一大优点。

支持这种函数式编程的语言有很多。受到数学强烈影响的函数式编程语言大多很难走出象牙之塔，下面介绍其中有名而又实用的几种语言。

14.1.1 具有多种函数式性质的Lisp

不久之前，Lisp 被看成是函数式编程语言的代表。实际上也是，Lisp 的基础是 Church 先生的 lambda 演算¹，函数本身也作为数据来处理，具有函数的第一级性质，并因此支持以函数为参数的高阶函数，因此 Lisp 具备函数式语言的很多性质。

¹ lambda 演算是指把函数定义和执行抽象化的计算模型。

下面来看一下用 Lisp 编写的阶乘计算程序。除了括号比较多之外，总体上讲与 Ruby 采用函数式编程方式编写的阶乘计算程序非常相似。

```
(defun fact (n)
  (if (= n 1)
      1
      (* n (fact (- n 1)))))
```

Lisp 最大的特征是 S 式记法。S 式是括号很多的 Lisp 的记法。初学者都非常讨厌 Lisp 的括号，但习惯了之后，就会发现它的优点。S 式无与伦比的优点是它彻底的统一性。也就是说，对 Lisp 而言，不管什么都可以统一成的单一形式。其他语言只是在函数调用时采用这种形式，而 Lisp 的函数定义、数据结构定义、算式以及流程控制等，全都采用这种形式。虽然也有人讨厌 Lisp 括号太多，但只要正确地处理好缩进对齐，这实际上并不是什么大问题。如果使用像 Emacs 一样支持 S 式的编辑器，它能找出对应的括号，自动调整缩进。

(函数 参数)

第二个重要之处在于链这种数据结构。Lisp 语言本名是 List Processor（链表处理器），这从一个侧面也说明了链的重要性。

链是一种构成树结构数据的通用数据结构，构成数据结构的节点分别包含两个引用。

Lisp 把节点称为 cons 单元，第一个引用称为 car，第二个引用称为 cdr，其中 cons 这种叫法来自生成新单元的函数 cons（construct 的省略）。因为最初的 Lisp 处理器把 car 数据放在地址寄存器，把 cdr 数据放在数据寄存器，所以 contents of address register（地址寄存器内容）省略为 car，contents of data register（数据寄存器内容）省略为 cdr。

另外，cons 单元以外构成链的数据元素（符号、数值等）称为原子（atom）。所以，链就是 atom 和 cons 单元构成的树型数据结构（图 14-2）。

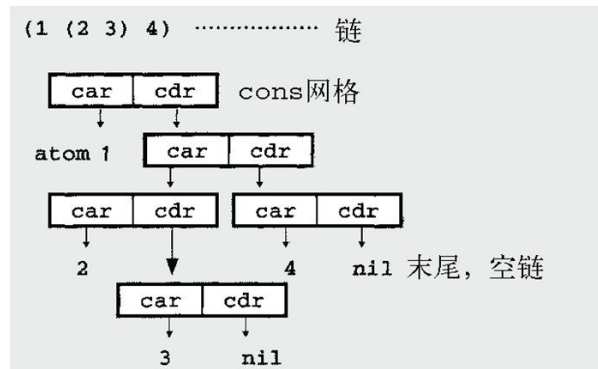


图 14-2 Lisp 的链

单看 S 式的话，感觉链好像跟数组是一样的，实际上它是像用环链接在一起的递归数据结构。这种递归数据结构非常适合于函数式编程，在 Lisp 以外的函数式编程语言中也得到广泛的应用。

函数式编程常常通过自我调用来实现递归调用，利用这种方式可以很自然地处理链这样的递归数据结构。

比如，让我们来定义一个链的求和函数 `list-sum`（参见图 14-3）。`list-sum` 首先取出链的头一个数据（`car`），然后对剩下的数据（`cdr`）同样调用 `list-sum` 函数，最后再求和。链这种数据结构是递归的，对应的函数调用也是递归的，与数据结构相对应的算法非常自然。

```
(defun list-sum (x)
  (if (null x)
      0
      (+ (car x) (list-sum (cdr x)))))
```

图 14-3 对链进行求和的 `list-sum`

综上所述，Lisp 具备了函数编程语言的基本特征。但是另一方面，Lisp 是一种历史悠久的语言，不直接支持多数纯粹的函数式编程语言所拥有的副作用²如回避以及延迟计算等功能。它对变量的赋值没有任何限制，习惯了结构化编程的程序员（比如我）在编写 Lisp 程序的时候，因为括号太多，很难适应函数式编程的风格。当然各有各的选择，这也不是什么坏事。

2 指因为某种处理而使程序状态发生变化。结果是，同样的输入却不能产生同样的输出。

近年来，由于像 **Haskell** 这样的彻底的函数式编程语言的兴起，**Lisp** 作为函数式编程语言在人们心中的印象变得越来越淡薄。下面就来介绍 **Haskell**。

14.1.2 彻底的函数式编程语言Haskell

Haskell 可以说是纯粹的函数式编程语言。也就是说，语言定义的本身与函数式编程密切相关，**Haskell** 编程与函数式编程是不可分割的。使用 **Haskell** 进行非函数式编程也不是完全做不到，但确实是非常困难的。

Haskell 是用数学家和逻辑学家 **Haskell Curry** 的名字命名的。用人名来命名编程语言也是很有传统的（比如：**Pascal**、**Ada**、**Occaml** 和 **B** 等）。**Haskell Curry** 先生对函数的应用很有研究，他的组合论与 **Alonzo Church** 先生的 **lambda** 演算几乎是一样的。另外，函数的部分应用“**Curry** 化”也是用他的名字来命名的。

Haskell 语言有如下特征：

- 没有副作用；
- 高阶函数；
- 函数部分应用；
- 延迟计算（非正式）；
- 静态多态类型系统；
- 型推论；
- 链内包表达式；
- 用对齐来表示块。

最后两个是纯粹的语法问题，其他特征都与函数式编程风格密切相关。

下面对 **Haskell** 的主要特征加以说明。首先，**Haskell** 程序基本上没有副作用，也就是说，不但不能改变变量的值，就连链的元素也是不能改变的。从某种意义上说，这是一种非常简洁的编程风格。

但是，如果不能改变变量的值的话，那程序该怎么编写呢？

首先，**Haskell** 程序中看起来像是变量的名字，实际上不过是函数参数或是值的别名。因此，根本就不存在作为保存变量值的变量。

Haskell 把其他语言中改变变量值的处理，都对应为重新生成一部分发生变化的新数据。因此，**Haskell** 没有赋值，只有给式命名，称为绑定。

没有变量，任何值都不会发生变化，因此，只要参数不变，函数的返回值也就总是一样的。相同输入总能返回相同结果，这种可重复性对测试程序非常有利。这种重复性也称为引用透明性。它和下面的静态多态类型系统结合起来，使 **Haskell** 这种函数式编程语言比其他的编程风格更难以出错。

Haskell 的第二个特征是高阶函数把函数本身作为数据来处理。比如，**Haskell** 中没有循环控制结构，循环都是用递归调用或者高阶函数来实现的。

14.1.3 延迟计算：不必要的处理就不做

第三个特征延迟计算的意思是必要的时候才进行处理。这个说法不够直观，让我们来看看实际的程序。图 14-4 中的 **Haskell** 程序类似于 UNIX 的 **head** 命令，从标准输入读取数据，只把前 10 行输出到标准输出设备中。

```
main = do cs <- getContents
          putStr (headLines 10 cs)
headLines n cs = unlines (take n (lines cs))
```


图 14-4 使用延迟计算的 Haskell 程序

这里对 Haskell 的语法不进行详细说明，只对构成程序的 Haskell 函数略加讲解。

main 是程序的入口点。执行程序的时候，首先执行 **main** 函数，这与 C 很相似。**do** 意味着按照顺序执行相同缩进水平的表达式。**getContents** 把标准输入的数据全部读进来。**lines** 返回按行分割后的字符串列表。**take** 是从列表中取出前 **n** 个元素。**unlines** 是把字符串的列表合并成一个字符串。

简单地说，图 14-4 中程序的意思是，读取标准输入的全部数据并显示前 10 行数据。但是，这仅仅是程序的意思，实际执行时的动作并不一定完全一样。

例如，当从标准输入来的数据非常庞大的时候，该怎么处理呢？**getContents** 表示把标准输入的数据全部读进来，按照一般的想法，它会把非常庞大的字符串全部读进来，然后按行进行分割，把前 10 行取出来之后，再合并成一个字符串并输出。前 10 行以外的数据读进来就扔掉，根本没有进行任何处理，这是很大的浪费。

试着用上面的程序来读取一个 192 万行、62M 的文本数据。令人吃惊的是，程序的执行瞬间完成，也几乎没有占用什么内存。

实际上在用 Haskell 对值进行计算的时候，只进行必要的处理，**getContents** 仅仅从标准输入读取 **take** 要处理的前 10 行数据，而并没有读取其他数据，这就是延迟计算。

与一般的编程语言不同，具有延迟计算性质的编程语言在字面上的意思与实际处理是不一样的。不进行多余的处理，即使是无限连续的数据也能处理，这是这种语言的优点。另一方面，程序的语句与实际的处理不是直接对应的，一旦出点儿什么问题，很难判断问题出在什么地方，这是这种语言的缺点。

14.1.4 灵活的“静态多态性”类型系统

与没有类型的 Ruby 语言不同，Haskell 是在编译时确定所有变量类型的静态类型语言。但实际上 Haskell 程序基本上没有必要给出变量的

类型。这是因为聪明的 **Haskell** 语言处理系统会推测变量的类型。

Haskell 的类型系统不是像 **C** 那样的不完全类型，而是健全的。因此，**Haskell** 程序如果编译不出错的话，那就证明该程序的类型系统没有问题。这时，**Haskell** 程序也绝对不会发生因为类型不匹配而异常中断的事情。还有，因为类型与算法的本质是密切相关的，类型检查可以排除大量的程序错误。

静态类型因为缺乏灵活性而受到批判。像 **Java** 等，仅仅接受预先声明的类型，如果事前准备不足、接口定义不够的话，很难应对将来的变更。

动态类型语言不进行这样的类型检查，显得非常宽容，只要拥有相同的方法，什么对象都可以，这就带来了柔软性，这种风格称为 **duck typing**。

Haskell 利用多态性的类型系统和类型推测，在维持着接近于 **duck typing** 灵活性的同时，也使编译时的完全类型检查成为可能。但是，如果类型推测出点儿问题，**Haskell** 程序出了类型错误的话，就会出现令初学者感到恐惧的错误。

14.1.5 近代函数式语言之父OCaml

OCaml 也是具有代表性的函数式编程语言之一。它比 **Haskell** 还古老，说它是近代函数式语言之父也毫不为过。

OCaml 起源于法国，在欧洲很受欢迎，**O'Reilly** 最早出版的讲解 **OCaml** 的书籍就是法语版的。即使没有这些，函数式语言在法国也是备受欢迎的，也许是因为喜欢数学的人多吧。

与 **Haskell** 相比，**OCaml** 具有如下的不同之处：

- 有副作用；
- 没有延迟计算；
- 具有强力的模块系统。

同样是函数式语言，前面两点表明了 **Haskell** 和 **OCaml** 设计思想的不同。

当然，**OCaml** 也并不推荐使用赋值以及改变变量值等具有副作用的计算，但与从语言上禁止副作用的理论派 **Haskell** 相比，**OCaml** 给人以实用派的印象，在必要的时候不惜牺牲一些理论，允许具有副作用的计算，延迟计算也是这样。如果想用 **OCaml** 来进行延迟计算的话，需要明确地进行延迟处理。

虽然不像 **Haskell** 那样自然地做到延迟计算，但换来的好处是，程序内容与实际处理关系接近，使人容易理解程序的执行过程。

14.1.6 强于并行计算的Erlang

Erlang 也是函数式编程语言，最近越来越受到关注。作为函数式编程语言，**Erlang** 具有如下令人回味的两个特点。

- 受到 **Prolog** 的影响
- 专用于并行计算

提到 **Prolog**，它最早是在第 5 代计算机研究中受到人们关注的理论型编程语言。虽然两种语言都有很强的数学背景，但函数型语言的诞生受到理论型语言影响，这对我来说还是一件意外的事。也许对于具有数学素养的人而言，这并不是什么令人吃惊的事。

与 **Haskell** 和 **OCaml** 不同，**Erlang** 没有类型（动态类型），也没有延迟计算，这也是令人觉得有趣的地方。

Erlang 的存在意义就在于并行计算，这种说法毫不夸张。**Erlang** 的基本是基于 actor 理论的并行计算。如果不使用 actor（**Erlang** 术语是过程）的话，**Erlang** 基本上无法进行任何处理。

Erlang 是针对并行计算而设计的，这与它是函数式编程语言也不无关系。甚至可以说，就是因为要进行并行计算，所以才选择了函数式编程风格。函数式编程没有副作用。改变数据时的同步处理是并行计算中最麻烦的部分，而如果根本就没有赋值、根本就不去改变数据的

话，同步处理也就没有必要了。Erlang 程序就不必担心数据访问的同步问题。

就今后 CPU 的发展方向而言，不能再指望单个 CPU 的处理速度飞速提高，理所当然是在一台计算机上安装多个 CPU，朝多核方向发展。在多核时代，软件也需要适应多核，但对我们来说，并行编程还是一个非常困难的领域。以 Erlang 为起点的函数式编程语言，没有副作用也不需要同步，具有非常适合于并行计算的性质。

14.1.7 用Ruby进行函数式编程

那么让我们来看一下，如果用面向对象的编程语言 Ruby 来进行函数式编程的话，应该怎么做呢？

从函数式编程观点来看 Ruby 的话，有几点需要留意的地方。首先，最重要的一点就是 Ruby 没有函数。

Ruby 的各种处理全是方法，也就是说任何处理总是和某个对象紧密地结合在一起。比如，即使是输出的 `print`，它也是 `Object` 类的方法，只是因为所有的类都是从 `Object` 继承的，所以可以省略调用的对象，使用时看起来像函数一样。

这就麻烦啦。这么说来，难道 Ruby 是不可能进行函数式编程？

哎，没有这样的事。当然 Ruby 最初不是作为函数式编程语言而设计的，不可能像 Haskell 一样来进行函数式编程，但应用 Ruby 也完全能够实现函数式编程的精髓。Ruby 中有几个能进行函数式编程的工具。

Proc对象（lambda）

Ruby 中唯一与函数直接对应的是 `Proc` 对象。换个说法就是，`lambda` 方法返回的是对象化的处理块。`Proc` 对象的 `call` 方法执行块的处理。

在 Ruby 1.9 中，可以用如下的方式进行调用（注意括号前面有一个点）

```
proc.(arg)
```

这种处理很类似于其他语言的函数调用。

程序块

以程序块为参数的方法等价于函数型语言的高阶函数。也就是说，灵活运用以程序块为参数的方法，就可以实现函数式编程的高阶函数的技巧。

进而在 Ruby 1.9 中，不用 `lambda` 方法，而用如下的语法

```
->(args){...}
```

也可以得到 **Proc** 对象。随着函数式编程风格的普及，预计到使用 **Proc** 对象的机会会越来越多，所以增加了这个比 `lambda` 还简洁的表达式。

枚举器

枚举器是从 Ruby 1.8.7 开始正式引入的，它的 `each` 方法可以循环返回对象。Ruby 没有延迟计算，但使用枚举器对数组和列表进行循环，可以实现类似于延迟计算的处理。

避免副作用

面向对象的编程中，副作用是几乎无所不在的，调用方法来改变对象的状态完全是日常工作，毫不为奇。但即使这样，避免副作用也是大有益处的。

所谓避免副作用，就是对生成的对象，尽量少去改变其状态。仔细想一下就会明白，我们在调试程序的时候，总是在程序里到处加入 `print` 语句，或者用调试器检查数据，其最大的原因就是 we 不知道数据当前的状态。

但是，如果不改写数据的话，它就总是处于相同的状态。因此，数据状态变得不明白的风险就会剧减。

具体说来，避免调用改变字符串或数组内容的方法（比如 `gsub!`、`<<` 等），设计类的时候避免改变实例变量的方法，都是行之有效的方针。

即使不局限于使用函数式编程语言，减少副作用也是很有益处的。Ruby 的群发邮件中有一个初学者常常容易被绊住的例子（参见图 14-5）。

```
• 生成有三个元素的数组
a = ["abcd"] * 3
p a # => ["abcd", "abcd", "abcd"]

• 改变第一个元素的内容
s = a[0]
s.upcase!
p s # => "ABCD"

• 其他的元素也随之发生变化
p a # => ["ABCD", "ABCD", "ABCD"]
```

图 14-5 关于副作用的例子

图 14-5 的程序首先生成一个有 3 个元素的数组，然后改变其第一个元素（字符串）的内容。但在这个时候，检查一下数据就会发现，本来只想改变第一个元素，结果是数组中所有元素的值都被改变了。

初学者看到数据被出乎意料地改变，往往会怀疑“这不是 Ruby 的程序错误吧”。实际上因为数组各元素引用的都是同一个字符串对象，所以改变第一个元素也会同时改变数组中其他元素。Ruby 程序通过引用构建了对象的网络，如果不留心的话，改变对象的内容会带来意想不到的影响。

14.1.8 用枚举器来实现延迟计算

从 Ruby 1.8.7 开始，很多以块为参数的方法在参数不是块的时候，就会返回枚举器。

枚举器就是把循环用对象来表达的一种方法。例如，从 `each` 方法返回的枚举器，就可以把传递给 `each` 方法的值按顺序取出来（参见图 14-6）。

```
e = [1,2,3,4].each
loop do
  p e.next
end
# 输出:
# 1
# 2
# 3
# 4
```

图 14-6 关于枚举器的例子

使用这样的枚举器可以实现与 **Haskell** 类似的延迟计算。与 **Haskell** 的列表不同的是，因为 **Ruby** 本身并不具备延迟计算的功能，所以对于数组的求值是一下子同时求出数组中所有的元素。

下面这一行程序是与图 14-4 的 **Haskell** 程序进行同样处理的 **Ruby** 程序。

```
puts ARGF.readlines.take(10)
```

初看起来，它比 **Haskell** 版还要短许多，但这个程序是把标准输入的数据全部读进内存的，如果传递的数据非常大，那么占用内存就会非常大，执行时间也会变得很长。

程序中出现的 **ARGF** 是个虚拟文件，代表标准输入或者用参数指定的文件输入。**ARGF** 的 **readlines** 方法读取 **ARGF** 所代表的虚拟文件，返回字符串数组。**take** 方法取出数组前头的元素（这里是前 10 行）。**take** 方法返回数组，该数组又作为 **puts** 方法的参数，照原样打印到标准输出上。

与 **Haskell** 版相比，**ARGF** 本来就是以行为单位进行处理而设计的，**puts** 会把字符串数组原封不动地输出，因为有了这些不同，程序变

得非常简洁。

这里的问题在于 **readlines** 方法会一下子读取文件的全部内容。实际上让这个程序读一下传递给 **Haskell** 版的同一文件时，执行时间是 2.19 秒。这里如果能够实现延迟计算的话，程序就会像 **Haskell** 版一样不进行多余的读取处理，程序的执行就会在一瞬间完成。

为此，我们把一下子读取全部文件内容并返回字符串数组的 **readlines** 方法替换成返回枚举器的 **each** 方法。使用 **each** 方法的程序变成下面的样子（实际上 **ARGF** 有 **take** 方法，本来是不需要调用 **each** 方法的，这样程序会变得更短，但为了使用枚举器，这里特意调用了 **each** 方法）。

```
puts ARGF.each.take(10)
```

因为传递给 **ARGF** 的 **each** 方法的参数不是块，我们就可以得到一个枚举器，它表示 **each** 应该返回各行数组。

对同样的数据来执行枚举器版程序时，这次的执行时间变成了 0.02 秒。程序执行速度提高了 100 倍。按照同样方法使用枚举器的话，**Ruby** 也可以实现对无限长列表的操作。

通过以上的例子，我想读者已经明白了，**Ruby** 虽然没有 **Haskell** 那样的默认的延迟计算，但如果灵活运用枚举器等工具的话，**Ruby** 也基本上能够很自然地实现延迟计算。

14.2 自动生成代码

置身于计算机行业，我们会强烈地感觉到日本很容易受到外国的影响。计算机技术差不多都来源于外国，这也是无可奈何的事。在美国制造的硬件体系结构的 **PC** 上，使用美国开发的操作系统（我使用的是芬兰开发的），使用的应用程序也大都是外国开发的。同样，日本软件的流行也总是落后于美国半年或数年之久，只要了解一下海外的情况，大致就可以预测日本软件行业的未来。

纯粹由日本发明的 Ruby 也有同样的倾向。先是 Ruby on Rails 框架在世界上得到了广泛的关注，然后日本国内也开始发生变化，以软件开发为生的人也开始关注 Ruby。以往总是使用 Java 或 PHP 的程序员也开始考虑用 Ruby 来开发工作上的软件。以前我曾经听到过有人说“虽然学习了 Ruby，但在工作中却没有使用的机会”这样的话，现在又感觉到日本也终于进步了。但同时令我痛心的就是 Ruby 在日本的普及也不例外，还是落后于海外。

Ruby 终于在商业领域也有了用武之地，这让我感到由衷的高兴，同时也在考虑如何更好地灵活应用它。

14.2.1 在商业中利用Ruby

2002 年在华盛顿州的西雅图举行了 Ruby Conference 2002，在这个会议上，超级程序员 Andy Hunt 先生介绍了在工作中应用 Ruby 的 4 个阶段。

1. 作为兴趣来学习的 Ruby。
2. 作为个人工具的 Ruby。
3. 工作中作为辅助开发工具的 Ruby。
4. 发布程序的 Ruby。

第一阶段是作为兴趣来学习的 Ruby，就是对 Ruby 有所关注，开始学习的阶段。通过读书、运行例子程序以及编写小程序积累点经验，了解 Ruby 这门语言及其背后的设计思想，就属于这个阶段。当然不管是谁都是从这里开始的。

第二阶段是作为个人工具的 Ruby，是指在日常生活中使用 Ruby 作为工具的阶段。比如，整理数码相机照片的小程序；每天监视日志文件，发现异常时发邮件通知的程序。使用 Ruby 的话，可以很简单地编写这样的小程序，实际上我以前也编写过不少这样的程序，现在平常还在使用。

第三阶段是工作中作为辅助开发工具的 Ruby，到了这一阶段，虽然发布的程序不是用 Ruby 编写的，但开始使用 Ruby 编写辅助开发工作的

工具。

几乎所有的软件开发应用都是团队工作，编写程序的开发语言几乎都是根据领导或客户的意见来决定的。在这种情况下，选择 C++、Java 或者 PHP 的情况就会比较多。

但是，关于辅助工具的开发，大都可以由开发人员自由选择。测试工具、文档生成工具、辅助构建工具、统计工具以及这次要讲解的代码自动生成工具等，都是有代表性的辅助工具。

最后是喜欢 Ruby 的人理想的最高阶段，发布程序的 Ruby 阶段。几年前除了非常有限的环境，这还是像是说梦话，但近来随着 Ruby 知名度的提高，Ruby on Rails 的强劲发展，从 2006 年前后开始，特别是在 Web 应用程序领域，这已经不再是珍闻了。

亲爱的读者朋友，你处在什么阶段呢？可能大多数都还处在第一或者第二阶段。但是，形势在急剧发展变化中。

14.2.2 使用Ruby自动生成代码

在上面介绍的 4 个阶段中，本节内容属于工作中作为辅助开发工具的 Ruby 阶段，具体讲解代码生成（Code Generation）的技巧。

代码生成就是字面所述的生成程序代码的意思，也就是说，利用某种模板编写程序，让它来自动生成实际的应用程序。

对于像 Ruby 这样的动态语言，代码生成技术并没有什么很大的效果，这样的语言本身具有丰富的处理程序的元编程功能，并不需要依靠代码生成，就可以实现对程序本身的操作。

说起来生成程序的编程，也许有很多读者朋友会感到不知所云。也许会有人认为：我想象不出在 Java 编程中有什么地方可以引入代码生成。

为此，让我们首先来看几个例子，代码生成在其中发挥了重要作用。

大家在编程的过程中，应该会感觉到有时候总是在重复编写相同内容的代码。编程中的代码重复是非常恶劣的。发现了代码重复，就应该

考虑在合理的代价范围内避免代码重复的方法。

话虽然是这样说，但并不全都仅仅是抽出共同方法这种情况。像 Java 这样的语言，因为没有 C 语言的宏定义功能，有些代码重复的情况实在是无法避免的。

14.2.3 消除重复代码

请看一下图 14-7，这是用 C 来编写的 Ruby 源代码的一部分。我们注意到，先是有个定义方法的函数，然后在相距非常远的地方，又有一个登录该方法的例程。像这样的话，仅仅定义了函数，却忘记了登录，或者反过来，忘记定义函数了，这都不是什么令人奇怪的事情。不单是有这种可能性，在实际编程中，发生这种错误的现象屡见不鲜。只要有发生错误的可能性，这种错误就一定会发生，这就是程序员。

```
static VALUE
rb_str_length(VALUE str)
{
    return LONG2NUM(RSTRING(str)->len);
}

...

void
Init_String(void)
{
    ...
    rb_define_method(rb_cString, "length", rb_str_length, 0);
    ...
}
```

图 14-7 Ruby 源代码示例

这里如果应用代码生成技术的话，仅仅给定义方法的函数添加些附加信息，就会自动生成初始化例程。比如像图 14-8 这样的写法。

```
/* def String#length(0) */
static VALUE
rb_str_length(VALUE str)
{
```

```
return LONG2NUM(RSTRING(str)->len);  
}
```

图 14-8 灵活应用代码生成的 Ruby 源代码示例

看一下图 14-8，我们会明白，仅仅是增加一行注释语句，函数的登录部分就可以完全省略不写。图 14-8 中省略了函数 `Init_String()`，实际上代码生成工具会生成与之相当的部分。为了在构建程序的时候，自动生成省略部分，我们只需要定义 `make` 的规则，在构建程序的过程中调用代码生成工具。

在对象语言没有元编程功能，或者元编程功能不强的情况下，代码生成最有效果。具体而言，在使用 C、C++ 或 Java 等语言的项目中，代码生成大有用武之地。

代码生成并不是 Ruby 所特有的技术。实际上，有很多用 Java 来生成 Java 的代码生成工具。比如，O/R 映射¹ 的 `Hibernate` 就属于这样的自动生成工具，它用 Java 自动生成数据库访问类的代码。还有，`XDoclet` 也是自动生成工具，我们只要给实体 `Bean` 类附加一些 `JavaDoc` 形式的注释，`XDoclet` 就会读取这些信息，自动生成 `EJB` 框架所需要的大量文件（会话 `Bean`、无状态 `SessionBean` 等）。

1 O/R 映射（Object/Relational mapper）是自动把对象和关系型数据库的表格对应起来的软件。

14.2.4 代码生成的应用

作为本节内容的参考，我推荐全面讲解代码生成的书籍 *Code Generation in Action*。这本书用英文介绍了代码生成在各种领域中的应用。这本书封面上并没有 Ruby 的字样，但代码生成的所有程序都是用 Ruby 编写的。

根据 *Code Generation in Action*，代码生成技术的应用有如下几个方面。

数据库访问

从数据结构定义自动生成数据库访问例程（包括 SQL）。比如在使用 EJB 框架的场合，每一个表都需要生成好几个文件，总的文件数有几百甚至上千个，也都不是什么稀奇的事。

手工编写大量的文件当然不是聪明人应该做的事。代码生成可以帮助我们解决这个问题。

用户接口

大多数依赖于数据库的 Web 应用程序，常常需要对表的各项项目进行 CRUD 操作。CRUD 是 Create（新建）、Read（读取）、Update（更新）和 Delete（删除）的首字母缩写。在极端的情况下，如果一个 Web 应用程序有 10 个表，就需要编写 10x4 种不同的 CRUD 画面。在这种场合，代码生成也有自己的用武之地。Ruby on Rails 最初也使用了同样的代码生成。

代码生成并不仅仅适用于 Web。Code Generation in Action 还介绍了 Java 的 GUI 工具箱 Swing 以及 MFC（Microsoft Foundation Classes）的对话框自动生成的例子。

单元测试

代码生成对单元测试也很有效果。只要有数据和测试条件，就可以开始单元测试，但实际上开始测试时，需要编写很多繁琐的代码。使用代码生成技术，雷同部分的测试代码就可以放手自动生成，而集中精力搞好测试条件。

对于使用数据库的应用程序，在测试之前准备合适的测试数据也是一件非常麻烦的事情。代替这种数据库 Mock 对象代码，从正式运行环境数据库提取出测试用数据的数据装载程序，其代码也都是代码生成的对象。

客户界面

在需要往系统里追加 XMLRPC、SOAP 或者 REST 等 API（应用程序编程接口）的场合，接受 API 请求的入口部分（接受请求，分析数据，调用实际处理）的代码都是固定模式。这里也有应用代码生成的余地。

文档化

代码生成也适合于文档化。文档本来不是代码，也许严格讲不能称为代码生成。

对于自动生成的类，由人工记入文档是纯粹的浪费，当然应该在生成代码的同时也自动生成文档。按照 JavaDoc（Ruby 有 RDoc）的形式，在生成代码的程序里写好相应的文档，应该是个聪明的方法。

14.2.5 代码生成的效果

代码生成对负责开发的工程师和以管理为主的经理都大有裨益。对工程师而言，代码生成有如下的好处。

- 改进质量。
- 确保一致性。
- 集中知识。
- 增加用于设计的时间。
- 独立于程序实现的设计判断。

能够灵活运用代码生成技术的项目，也都是事出有因的。比如，表的个数很多，要生成大量雷同的窗口（页面），这就会带来大量的重复代码，这也就是灵活运用代码生成的条件。

用代码生成工具来统一处理程序代码，就可以避免代码的分散，把重复集中到一个地方，也就是集中到对工具的输入，或者集中到工具本身。

这样就可以改进质量，确保大量雷同代码间的一致性。因为类似的代码是由工具一下子生成的，发生变化的时候，要修改的地方也是有限的，有希望提高生产率和可维护性。

进一步讲，要开发代码生成工具的话，首先就会在一开始的时候，充分研讨什么地方是类似的，什么地方是不同的，这样就可以避免毫无

章法的开发。因为有了这样的抽象化要求，就不会迁就个别程序实现的特殊要求，可以集中于软件应该有的样子而进行设计。

不管怎么说，合适地运用代码生成，就更有可能专心于正确的设计工作，**Jack Herrington** 先生就是这样说的。

另一方面，负责管理的经理也可以从代码生成获得如下的好处。

- 提高成本预测的可能性，简化管理。
- 提高产品质量。
- 维持士气。

采用具有一致性的框架，不仅给开发人员带来容易开发和维护的优点，经理也可以从中获得容易预测成本和管理开发人员的好处。代码生成能够实现的代码高质量对经理来说也是件令人高兴的事。项目具有能够更强地适应变化的倾向，也就更有可能进行小而快的敏捷开发²。最后，对于开发人员来说，因为能够维持舒适的环境，就有可能保持开发人员的士气。

² 敏捷开发是具有小而快特征的软件开发方法论，相对于过程与工具，这种开发方法更着重于个人的相互作用。

但是，代码生成并不是什么灵丹妙药。仅仅是引入代码生成并不能解决项目所有的问题。

如果不充分分析要开发的软件本质，准备好高质量的代码生成器，项目就很难取得成功。也就是说，为了从项目中获得利益，事前的充分准备是十分重要的。

14.2.6 编写代码生成器

在看代码生成的实例之前，先来讲解一下能用于编写代码生成器的工具。

对代码生成最有帮助的工具恐怕要数 **eRuby** 了吧。**Ruby** 本身也拥有优越的文本处理功能，有幸被 *Code Generation in Action* 选中作为开发工

具的语言，是适合于代码生成的，然而使用 eRuby 的话，在以比较自然的形式编写模板的同时，还可以灵活运用 Ruby 的处理能力。

作为 Web 应用程序的模板，好像使用 eRuby 的比较多。但是，作为模板系统的 eRuby 本身并不依赖于 Web，也不是只能用来生成 HTML。与依赖 SGML 标签的 Amrita 模板系统相比，这是其显著不同的特点。

请看图 14-9，这是 eRuby 程序的例子。记号<% 和 %>括起来的部分是 Ruby 程序。记号外面是照原样输出的部分。以记号<%=开始的 Ruby 代码会把代码的执行结果嵌入到输出的文本里。

```
现在是<%= Time.now.to_s %>  
<% 3.times do |i| %>  
<%= i %>  
<% end %>
```

图 14-9 eRuby 程序示例<%%> 括起来的部分代表 Ruby 代码

要执行图 14-9 的程序，需要启动 eRuby 的处理程序³。使用 eruby 的场合，要执行如下的命令。

³ eRuby 是文本嵌入型语言的名称。eruby 与 erb 是处理命令的名称。简单而言，eRuby 就是允许在普通文本中嵌入 Ruby 的代码片段。

```
$ eruby list3.erb
```

erb 也一样。

```
$ erb list3.erb
```

两者的执行结果如图 14-10 所示。其中的空行是没有输出的程序部分消失之后所留下的痕迹。作为代码生成对象的编程语言大都不对空行作任何处理，这些部分也就可以忽略。


```
现在时间是Sat Jan 14 00:41:20 JST 2006
```

```
0
```

```
1
```

```
2
```

图 14-10 图 14-9 程序的执行结果，嵌入的 Ruby 代码被执行

在保存图 14-9 中的文件时，使用了`.erb`的扩展名，但实际上扩展名没有任何实际意义。

`eruby` 与 `erb` 都是处理程序，基本上是互相兼容的，只有如下 3 点不同。

1. `eruby` 是用 C 编写的，`erb` 则是用纯 Ruby 编写的。因此处理速度有时会有差异。
2. `eruby` 具有对应 CGI 的文件头输出功能，而 `erb` 则没有这样的功能。
3. `erb` 是 Ruby 的库，可以从程序中简单调用，而 `eruby` 则需要用命令来启动别的过程。

因为最后一个特征，使得`erb`更适合于代码生成。`eruby`是独立的程序，在Ruby之外需要单独安装⁴，而`erb`则是Ruby的标准库，这一点也是很有帮助的。

⁴ `eruby` 是与 `mod_ruby` 一起开发出来的，`mod_ruby` 中包含了 `eruby` 的功能。

从程序中使用`erb`的时候，程序代码如图 14-11 所示。

```
require 'erb'

template = ERB.new <<-EOF
现在时间是<%= Time.now.to_s %>
<% 3.times do |i| %>
<%= i %>
```

```
<% end %>
EOF
puts template.result
```

图 14-11 使用 erb 库的例子

首先，以模板的字符串为参数，生成 **ERB** 类的对象。然后，调用该对象的 **result** 方法来执行代码生成程序。如果想要在模板中访问局部变量的话，就需要在调用 **result** 方法的时候，传递给它保存有局部变量状态的 **Binding** 对象。调用 **binding** 方法可以得到 **Binding** 对象。

14.2.7 也可以使用XML

如果利用代码生成工具的开发团队的所有开发成员都对 **Ruby** 抱有好感的话，**Ruby** 本身就可以作为一种 **DSL**⁵，用来为程序生成器生成输入。如果情况允许的话，这是最简单的啦。**Ruby** 解释器本身会对输入进行解析，就用不着另外编写解析例程，也不需要再进行加工。习惯之后，可以完全抛弃烦琐的配置文件，非常简单地编写出代码生成器。

5 DSL (Domain Specific Language) 是指针对特定领域而强化功能的小规模编程语言，在第 8 章有讲解。

但是，开发团队中其他开发人员，经理或者客户可能会这样说：“选择用 **Ruby** 作为开发工具当然是你的自由，但总不能强迫其他开发人员来都来用 **Ruby** 吧。”这种场合，最有效的解决方法就是：“那么，用 **XML** 来作为输入吧。”

虽然不清楚详细原因，但是好像有某种法则，很多经理大都能接受“要是 **XML** 的话，那就可以吧”，这是很方便的事。我们的目的并不是要让经理也喜欢起 **Ruby** 来，而是为了使用代码生成来提高工作效率，所以在 **XML** 这一点上妥协也是应该的。

幸运的是，**Ruby** 拥有解释 **XML** 的标准库 **REXML**。**REXML** 提供操作 **XML** 的功能，这里让我们来看一下读取简单的 **XML** 文件，表示其内容的例子。图 14-12 是把配置文件设计成 **XML** 文件的例子。使用

REXML，用图 14-13 中 Ruby 程序读取配置文件，配置信息表示成 Python 风格（参见图 14-14）。

```
<beans>
<bean name="Employee">
<attributes name="name" type="String"/>
<attributes name="id" type="int"/>
<attributes name="age" type="int"/>
<attributes name="section" type="String"/>
</bean>
<bean name="Section">
<attributes name="name" type="String"/>
<attributes name="id" type="int"/>
</bean>
</beans>
```

图 14-12 作为配置文件的 XML 文件示例

```
require 'rexml/document'

doc = REXML::Document.new(File.open("conf.xml"))
doc.root.each_element("bean"){|elem|
  printf "class %s:\n", elem.attributes["name"]
  elem.each_element("attributes") {|attr|
    printf " %s:%s\n", attr.attributes["name"],
    attr.attributes["type"]
  }
}
```

图14-13 使用 REXML 的 Ruby 程序，读取图 14-12 的 XML 文件

```
class Employee:
name:String
id:int
age:int
section:String

class Section:
name:String
id:int
```

图14-14 图 14-12 与图 14-13 程序的输出结果

14.2.8 在EJB中使用代码生成

让我们使用以上的工具来生成 EJB 的 EntityBean（样）的源代码吧。输入与图 14-12 的 XML 文件完全一样。请看 sample.rb（参见图 14-15），这是 EntityBean 样 Java 源代码的生成工具。因为是例子，生成的 EntityBean 实际上是无法使用的，但读了程序大致可以明白这个过程。

```
require 'erb'
require 'rexml/document'

doc = REXML::Document.new(File.open("/tmp/conf.xml"))
template = ERB.new <<-END
/**
 * The Entity bean for <%= name %>
 * @ejb:bean name="<%= name %>"
 *          display-name="<%= name %>"
 *          jndi-name= "ejb/<%= name %>"
 *          local-jndi-name="ejb/<%= name %>Local"
 */
public abstract class <%= name %>Bean implements EntityBean {
  <% elem.each_element("attributes") {|attr|

    aname = attr.attributes["name"].capitalize
    atype = attr.attributes["type"]%>
    /**
     * @ejb:persistent-field
     * @ejb:interface-method view-type="both"
     */
    public abstract <%= atype %> get<%= aname %>();
    /**
     * @ejb:interface-method view-type="both"
     */
    public abstract void set<%= aname %>(<%= atype %> <%=
aname.downcase %>);

    <% } %>
  }
}

END
doc.root.each_element("bean") {|elem|
  name = elem.attributes["name" ].capitalize
  open("#{name}Bean.java", "w") {|out|
    out.puts template.result(binding)
```

```
}  
}
```

图 14-15 Bean 生成程序

执行这个程序之后，当前目录中会生成各个 **Bean** 的源代码文件。输入文件 `conf.xml` 仅有 12 行，生成的 **Java** 文件长达 80 行。

看了程序我们知道，程序中定义了 **EJB** 所需要的繁杂的 **Getter** 和 **Setter**，它们的定义包括 **JavaDoc** 注释。最近像 **Eclipse** 这样的开发环境，具有插入类似的固定模式代码的功能，但我感觉，考虑到针对个别项目的定制容易性与应用范围，代码生成具有更大的可能性。

已经好久没有接触 **Java** 程序了，这次为了写这本书，又读了些 **Java** 程序，和我平常使用的 **Ruby** 相比，实在是十分冗繁。当然 **Java** 有 **Java** 的优点，但要写的长读的多，确实是件不容易的事。**Java** 阵营的人也理解这一点，所以开发了像 **Eclipse** 这样的开发环境，像 **XDoclet** 这样的代码生成等辅助工具，利用它们来提高生产效率。

14.3 内存管理与垃圾收集

垃圾收集（**Garbage Collection**，**GC**）是一种管理程序使用的内存区域的方法。使用具有垃圾收集功能的编程语言或处理程序的话，程序中不需要编写内存管理的代码，也就是说不用编写代码来释放使用过的内存区域。

14.3.1 内存管理的困难

在垃圾收集普及之前，内存区域的取得和释放完全是程序员的责任。即使是现在，**C** 或者 **C++** 这些编程语言也还是没有垃圾收集功能。

随着程序的执行，会使用一些内存区域作为作业区。为记忆对象、字符串和数组等个别信息，都需要使用内存区域。没有垃圾收集的语言一般都提供有 **API**，利用这些 **API**，可以在需要时向操作系统申请并取得内存，使用过之后再吧内存区域返回给操作系统。

C 语言从操作系统取得内存的函数是 `malloc()`，释放内存的函数是 `free()`。 `malloc()` 和 `free()` 的使用方法见图 14-16。

```
/* 使用时要包含stdlib.h include */
#include <stdlib.h>

/* 申请1024 字节的内存区域 */
char *p = malloc(1024);
if (p == NULL) {
    /* 发生错误时返回NULL */
}

/* 使用完之后 free () */
free(p);
```

图 14-16 C 语言中 `malloc()` 和 `free()` 的使用方法，程序必须明确调用 `free()`

在 C++ 中，为对象申请内存区域时使用 `new` 运算符来代替 `malloc()`，释放内存领域时使用 `delete` 运算符来代替 `free()`（参见图 14-17）。与使用库函数来实现内存区域管理的 C 语言相比，面向对象的 C++ 语言把对象管理融合到了语言本身。

```
class Foo {
    ...;
}

// 使用new 运算符来申请内存
Foo *p = new Foo();
//错误时会发生异常，所以不需要检查返回值

//使用完之后delete
delete p;
```

图 14-17 C++ 语言中 `new` 和 `delete` 的使用方法，与图 14-16 相比，取得内存区域时的错误处理变得简单了

像这种“手工”管理内存的方法，很容易发生各种问题。

悬挂指针

如果把还在使用中的内存区域错误地调用 `free()` 予以释放的话，就会发生悬挂指针（`dangling pointer`）的现象。在此之后，错误释放了的内存区域可能被程序用于别的目的，或者返还给操作系统。不管是哪种情况，它都会脱离程序管理，变成预想之外的状态。

这样，程序再去访问该区域的时候就会出错。在引用该区域内容的时候，多数场合会读取到被破坏的数据，或者是在更新该领域数据的时候，会置换成预想之外的数据。

内存泄漏

另一方面，如果忘记了把申请的内存区域返还给操作系统的话，就会发生内存泄漏（`memory leak`）。这时候因为使用过的内存区域越积越多，程序占有的内存就会逐渐增加。特别是长时间启动，连续提供服务的常驻内存型程序（`daemon`），容易发生内存泄漏，导致系统停止等重大问题。

二重释放

对已经释放过的内存区域再次调用 `free()` 的情况也时有发生。这称为二重释放（`double free`）。这会带来 `malloc()` 和 `free()` 内部使用的数据区域不一致的问题。

想要完全避免这些问题是非常困难的。特别是在面向对象的编程中，不仅存在大量对象数据，而且对象变得不需要的时刻也不明确。还有令人头痛的是，这些问题大都在犯错误的时候没有任何征兆，而在之后才会暴露出来。

内存泄漏问题只有在内存占用量超出预想的时候才会显露，而悬挂指针问题不会发生在释放使用内存的时刻，而是要到访问释放过的内存区域时才会出现。程序员往往会在看起来完全没有问题的地方突然遭遇意想不到的错误。

为检查出C或C++的内存问题，`valgrind`（<http://valgrind.org/>）或者 `electricfence`（<http://perens.com/FreeSoftware/ElectricFence/>）这样的内存分析器专用工具会起到很好的作用。使用这些工具，可以检查出以下问题。

- 试图对申请的内存区域之外的部分进行访问。
- 已经释放过的区域发生二重释放。
- 内存泄漏。

这些工具可以发现出问题的内存区域是在程序的什么地方申请的。但是，因为内存问题的特殊性质，仅靠这些信息是很难发现所有程序错误的。

14.3.2 垃圾收集亮相之前

应该管理的内存区域越来越多，内存释放时刻的管理也就越来越难，这是产生内存问题的原因。本来像这种大量对象的管理业务就不是人（程序员）所能做的。

人管理不过来的话，就应该放手让计算机来做。垃圾收集就是在这样的背景下诞生的。令人意外的是，其实垃圾收集早在 1960 年就呱呱坠地了，从现在说已经是 50 年前的事情了。当时编程语言 Lisp 也才刚刚诞生，Const 单元¹要生成大量对象，由人工来明确管理是不可能的事情。垃圾收集就是为解决这一问题而诞生的。

1 Lisp 的基本数据结构，是构成列表的节点。

从那以后，在 Lisp 与受到其影响的语言中，垃圾收集成为常识。但是，FORTRAN、COBOL、C 与 C++ 这些得到广泛使用的语言都没有利用垃圾收集。这是因为关于垃圾收集，有以下这些先入为主的观念。

垃圾收集慢

认为垃圾收集慢的观念只有一半是对的。计算机不可能完全理解人的意图。某内存区域是否不再使用，计算机不可能完全正确地判断出来。本来就是因为人不能判断才会发生内存问题的，跟计算机说“把所有不再使用的内存都找出来”也属于无理要求。

这样的话，就只能使用一些别的方法来找出不再使用的内存。比起人工直接管理内存区域的寿命，在不要的时刻再明确释放，这需要做些

多余的工作，所以人们都认为执行时间会变长。但是即使是在人工明确释放内存区域的场合，内存管理也还是需要一定开销的。有研究表明，在某些条件下，垃圾收集比手工管理内存还更快²。

2 Andrew W. Appel, “Garbage Collection can be Faster than Stack Allocation”, Information Processing Letters, vol.25, no.4, 1987。

垃圾收集可靠性低

垃圾收集可靠性低的观念可能是由于个别垃圾收集处理程序的质量不高而形成的。如果垃圾收集本身有程序错误的话，这些错误就可能导致前面列举的各种内存问题。实际上，我在 Ruby 的垃圾集中也有几次因为程序错误，而给大家带来了很大麻烦。

但是想一想的话，不使用垃圾收集的时候，也会经常发生内存问题，程序本身的可靠性随之降低，性能也就无从谈起。结果常常是程序员在无可奈何的情况下，只好自己想点办法来实现类似于垃圾收集的功能。与其每次独自实现垃圾收集，真不如利用已实现好的垃圾收集，这样才可以最终得到最大的好处。

Java 语言的存在打破了这两个偏见。1995 年登场的 Java 从一开始就具备垃圾收集功能，以后又经过持续不断的性能改善，扭转了垃圾收集慢得不能使用的偏见。

在 Java 之后诞生的编程语言，不管是否受到 Lisp 的影响，几乎都毫无例外地拥有垃圾收集功能。垃圾收集从诞生之日起，经过了 40 多年的发展，终于成为大家认可的一项特性了。

14.3.3 评价垃圾收集的两个指标

假如存在备有无限内存的计算机的话，垃圾收集是没有必要的。现实中内存容量是有限的，为了最大限度地利用有限的内存，就需要有垃圾收集。话虽然这么说，那毕竟只是受计算机资源的制约，这一点是不能忘记的。垃圾收集所进行的基本上都是用户看不到的处理，不能因为垃圾收集而过分影响处理性能。

从这一点出发，在评价垃圾收集算法的时候，一个重要指标就是尽量不要给用户带来影响，不要让用户等待。但是，没有任何算法能满足

所有人的要求，我们可以根据程序的性质来选择合适的垃圾收集算法。

垃圾收集的性能可以由以下两个指标来测定。

吞吐量

吞吐量（**throughput**）是垃圾收集处理在程序全部执行时间中所占的比例。当然，垃圾收集处理所占的比例越小越好，垃圾收集处理所占的比例大的话（换个说法就是吞吐量小），程序整体的性能就会低下。

暂停时间

暂停时间（**pause time**）是一次垃圾收集处理所中断的时间。暂停时间过长的话，处理的中断时间就会变得很显眼，程序的反应就会变慢。比如在射击游戏中，如果因为垃圾收集的处理而导致游戏机无法操作，并因此导致游戏结束的话，当然就会激怒玩游戏的人了。

垃圾收集每次所花的时间会因垃圾收集执行时对象的总数而变化，暂停时间的指标有两种，一是把垃圾收集时间求平均值所得到的平均暂停时间，二是程序执行中最长的垃圾收集时间所代表的最大暂停时间。

那些实时性要求高的程序，就很重视最大暂停时间。

吞吐量和暂停时间这两个指标都好的话当然比较理想，但往往很难达到。像批处理这样非对话的处理强调吞吐量要高，而重视反应速度的嵌入式控制或者 GUI 程序就强调暂停时间要短。

Java 运行环境中实现有多种垃圾收集算法，可以根据程序的性质来切换合适的垃圾收集算法。

14.3.4 垃圾收集算法

垃圾收集算法基本上是如下 4 类，还有几种变形。

- 引用计数方式。

- 标记和扫除方式。
- 标记和紧缩方式。
- 复制方式。

14.3.5 引用计数方式

引用计数方式在垃圾收集算法中具有最简单最容易实现的特征。与下面要介绍的标记和扫除方式并列，都是在早期（1960 年）发明的。

以下是它的基本原理。

首先，各对象知道自己被从几个地方引用着（被引用数、引用计数器）。然后，在引用增减的同时也相应地改变被引用数。引用计数器变成 0 的对象，也就明确地表明它不再被其他对象所引用，可以释放它所占用的内存区域（参见图 14-18）。

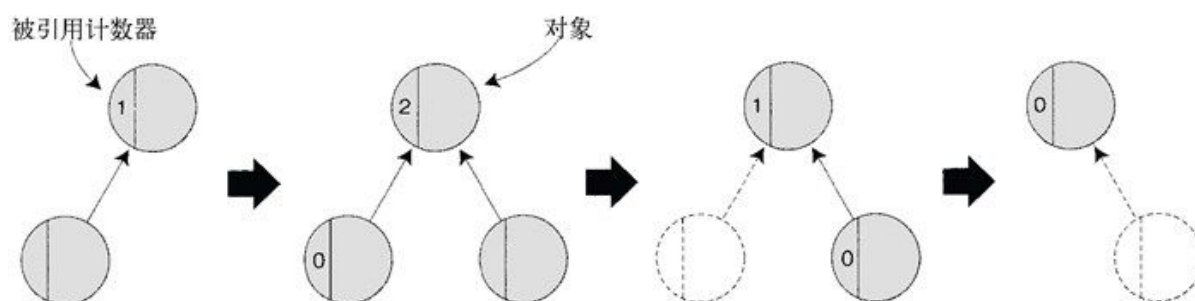


图 14-18 引用计数方式的原理，各对象内藏有被引用计数器，计数器变成 0 的对象将被释放

分析一个对象将来是否还会被用到，理解程序的意图来执行垃圾收集是不可能的。但是，如果一个对象不再被其他对象所引用，今后就确实不再会被用到。垃圾收集算法的基本功能就是找出这些不再被引用的对象。

引用计数方式的最大优点就是容易实现。这种方式得到广泛的使用，（包括我在内的）前几年的 C++ 程序员几乎都曾经实现过类似的引用计数器方式。

进一步讲，它最大的好处在于当引用数变成 0 的同时对象也就随之释放。在其他的垃圾收集方式中，当引用数变成 0 之后，对象什么时候

被释放，都还是一个未知数，而引用计数器方式可以同时进行释放处理。暂停时间短也是它的优点。

另一方面，它有 3 个缺点。最大的缺点是它不能释放有循环引用关系的对象群。图 14-19 中的对象 a、b、c 不再被外面其他对象所引用，但因为它们自己互相引用，引用计数器不会变成 0，这样的对象就永远不会被释放。

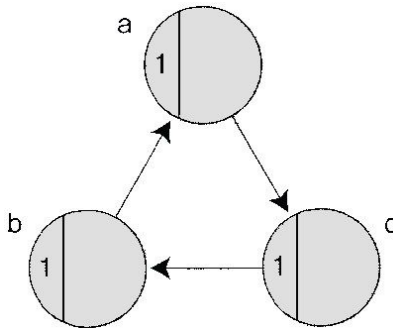


图 14-19 引用计数器方式无法处理的对象示例，因为有循环引用，引用计数器不会变为 0，会带来内存泄漏

引用计数器方式也有其与生俱来的缺点。因为在引用增减的时候有必要同时正确维护引用计数器的增减，忘了这一点就会带来悬挂指针或内存泄漏问题。在语言处理系统本身管理引用计数器的场合，还不太容易出问题，而程序员自己管理引用计数器的做法，没想到竟成为程序错误的温床。

最后 1 个缺点是，引用计数器的管理与并行处理不相容。如果多个进程同时增减一个引用计数器的话，就会发生引用计数器的值不一致的现象（从而成为内存故障的原因）。为避免这一问题，需要在操作引用计数器的时候进行排他处理，但在频繁发生引用操作的同时进行排他处理的话，会带来巨大的（时间）开销。

总之，这种原理简单实现也简单的引用计数器方式，缺点很多，最近已经不怎么用了。

现在，采用引用计数器方式的主要语言处理系统有 Perl 和 Python。为了回避循环引用问题，它们都组合了其他垃圾收集方式。这些语言基本上以引用计数器方式来进行垃圾收集，只有在极个别的情况下，才通过别的垃圾收集算法来处理引用计数器不能回收的对象。

14.3.6 标记和扫除方式

标记和扫除方式是和引用计数器方式同样古老的垃圾收集算法，相关论文都同样发表于 1960 年。

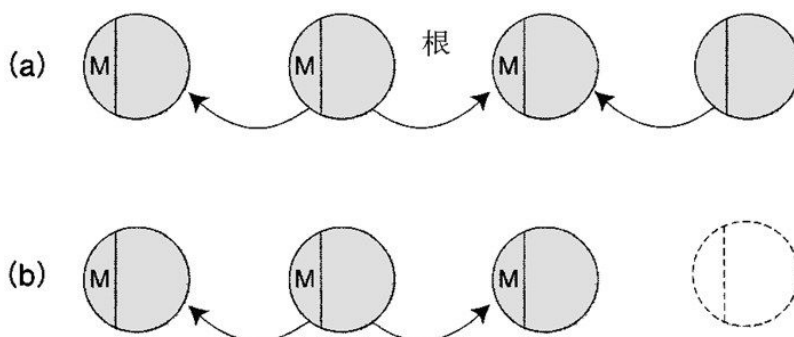


图 14-20 标记和扫除方式的原理。(a) 从根开始按顺序给所有被引用的对象加上标记 (M)；(b) 没有标记的对象被释放

标记和扫除方式从有可能成为对象引用元的变数（根）开始，给被引用的对象加上标记，表明它“活着”。这种方式然后再给标记对象所引用的对象，还有其再引用的对象，也都递归地加上引用标记（参见图 14-20a）。

这样从根开始不管是直接还是间接，只要把所有引用的对象都加上标记的话，那没有标记的对象就是没有被引用的，也就是“已经死了”。

然后，在所有的对象中找出没有标记的对象，把它们作为垃圾扫除出去（参见图 14-20b）。

这种标记和扫除方式虽然很古老，但确实非常优秀，现在也还被多种处理系统所采用。

但它也有缺点。当对象数目较多的时候，性能容易恶化。在标记的时候（标记阶段）要访问生存着的所有对象，在回收成为垃圾的对象时（扫除阶段），按顺序访问所有的对象，找出其中“已经死了”的对象。在寻找垃圾和扫除的过程中，基本上不能进行其他处理，垃圾收集时间长的话，会影响到本来的处理工作。

采用标记和扫除方式的语言有很多。我最熟悉的 **Ruby** 就是个代表性的例子。除了当对象数目多的时候有时会有问题外，大都执行得很好。

14.3.7 标记和紧缩方式

标记和紧缩方式是标记和扫除方式的变形。标记处理是一样的，后阶段有所不同。

标记和扫除方式按顺序检查所有的对象来进行扫除，标记和紧缩方式移动生存中的对象位置来腾出空间（参见图 14-21）。

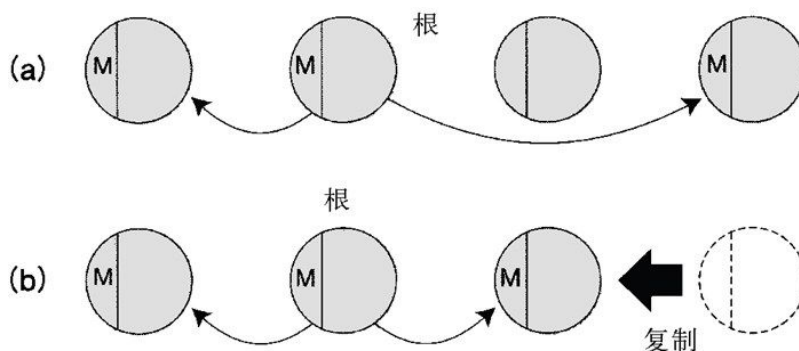


图 14-21 标记和紧缩方式的原理。(a) 对象的顺序虽然有所不同，标记处理与图 14-20 是一样的；(b) 移动生存中的对象，腾出空间

标记和紧缩方式最大的特征，也是它的优越之处，是把空间集中起来（紧缩）。紧缩的结果是把没有释放而活下来的对象都集中到了一个地方。这样，内存访问就集中到一个局部区域，这可以提高缓存功能的效率。对象的分配也只是把指针移动一下就完成了，降低了对象分配的开销，这也是它的好处。

这种方式的缺点是，把生存着的对象全部复制的紧缩开销，容易比标记和扫除方式中执行扫除的开销还要大。还因为对象被移动位置了，不能应用下文讲述的保守垃圾收集，这也是它的一个缺点。

一部分 Lisp 处理系统采用的是标记和紧缩方式，Java 处理系统（作为多个垃圾收集算法中的一个）也实现有标记和紧缩方式。

14.3.8 复制方式

像标记和扫除方式、标记和紧缩方式这样“标记之后释放死了的对象”的算法，标记时间与活着的对象数成比例，扫除（或者紧缩）时间与总对象数成比例。因此，在分配有非常多的对象，其中几乎所有对象都要被释放的场合，扫除的开销会变高，在性能方面很不利。与标

记时间一样，要是能够有一种算法，只需要与生存着的对象成比例的开销就可以回收内存区域的话，岂不是很好吗？

基于这种想法的算法是复制方式。复制方式与标记和扫除方式一样，从根开始扫描所有的对象，但它不仅仅是加标记，还执行复制（参见图 14-22）。

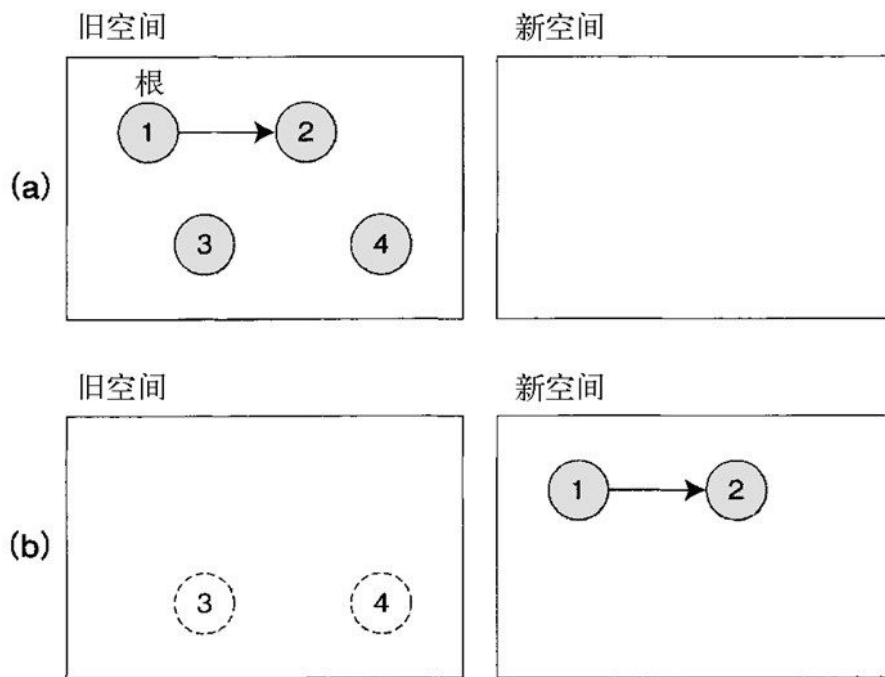


图 14-22 复制方式的原理。(a) 在旧空间里分配对象；(b) 旧空间填满的时候，从根开始

复制方式把内存空间分成旧空间和新空间两大块，总是在旧空间中分配对象。旧空间爆满的时候，从根开始扫描对象，把对象复制到新空间。这时，复制后的引用也要随之更新。

递归地执行从旧空间到新空间的复制，结束的时候就会把所有活着的对象都移动到新空间。不再被引用的对象都遗留在旧空间，旧空间就可以整个地弃之不用，也就避免了扫除的开销。从这之后再吧新空间作为这次的旧空间来继续同样的处理。

复制方式最大的优点是，垃圾收集开销只与活着的对象数成比例。与标记和紧缩方式一样，对象的分配开销低也是它的优点。

它还有“局部性”的优点。复制方式按顺序把引用的对象复制到新空间，关系近的对象会被分配在相近的内存空间上，这称为局部性。

我们知道，关系近的对象同时被访问的可能性也高。计算机上因为有缓存，内存空间相接近的访问有可能具有较好的性能，局部性高的程序具有提高性能的优势。

另一方面，复制方式也有缺点。复制方式要复制所有活着的对象，几乎具有与标记和紧缩方式同样的缺点。

一部分 Lisp 处理系统采用单纯的复制方式。还有，很多 Java 虚拟机把复制方式与其他方式组合起来，提供下文所述的分代垃圾收集。

14.3.9 多种多样的垃圾收集算法

垃圾收集的基本算法大致可以归结为以上 4 种，并在此基础上有各种变形。

此外，还有把基本算法组合起来的几种技术，这里介绍其中几个具有代表性的技术。

- 分代垃圾收集
- 保守垃圾收集
- 增量垃圾收集
- 并行垃圾收集
- 位图标志

这些技术的组合也是有可能的。

14.3.10 分代垃圾收集

首先来讲解垃圾收集技术中最有名最重要的分代垃圾收集（generational GC）。

分代垃圾收集的基本思想是利用程序和对象的性质。一般的程序都有这样一个性质，几乎所有的对象都在比较短的时间里变成垃圾，存活时间超过一定程度的对象总是拥有更长的寿命。

寿命长的对象更容易活下来，寿命短的对象会在更短的时间内变成垃圾，因为这一性质，就可以重点对分配之后还没有怎么经过时间的“年轻的”对象进行扫描，这样的话不用扫描全部对象就可能高效率地回收垃圾。

具体来说，分代垃圾收集把对象的内存空间分成两个，分别是容纳年轻对象的“新代”用的区域和容纳长寿对象的“旧代”用的区域。有的实现按 3 代以上进行划分，这里为了简单说明，只考虑两代的情况。

如果前面关于对象寿命的假设成立的话，仅仅扫描容纳年轻对象的新代空间，就可能以非常高的比例大量回收成为垃圾的对象，这种只扫描新代区域的回收称为轻垃圾收集。

但是，单纯地只对新代区域扫描，释放的算法是不能达到良好效果的，因为存在有旧代区域对新代区域的引用。

如果只扫描新代区域的话，就没有检查旧代区域对新代区域的引用，只被旧代区域引用的年轻对象就可能被误判为“已经死掉了”。这就会带来内存问题。

分代垃圾收集解决这一问题的办法是，监视对象的更新，在旧代区域引用新代区域发生的同时，就把这一引用的记录例程以及对象的更新场所全都记录下来，这个检查例程叫作写屏障（**write barrier**）。记录旧代区域对新代区域的引用称为记录集（**remembered set**）。

旧代区域中对象一般具有寿命比较长的倾向，但绝不是说它“总也不死”。随着程序的执行，旧代区域中也会积聚起垃圾。旧代区域中垃圾不回收的话，也会发生内存泄漏，所以需要不时地扫描包括旧代区域在内的所有区域，进行全面的垃圾收集。以所有区域为对象的垃圾收集称为全垃圾收集或重垃圾收集。

以上说明的分代垃圾收集的原理如图 14-23 所示。

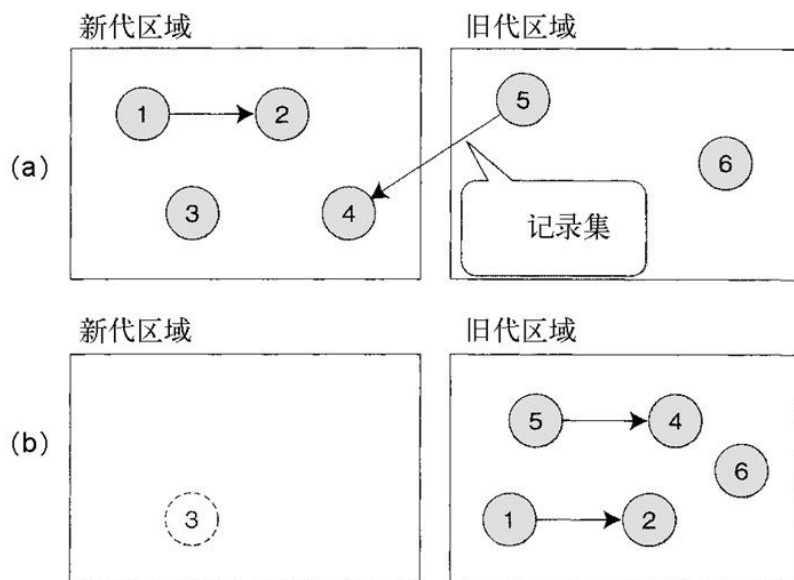


图 14-23 分代垃圾收集的原理。(a) 把寿命短的新代区域与寿命长的旧代区域分隔开来管理对象，用记录集来记录两者之间的引用；(b) 新代区域中有引用关系的对象移到旧代区域中，释放没有引用关系的对象

分代垃圾收集减少了扫描对象的个数，有缩短平均暂停时间的效果。但是，因为要执行全垃圾收集，最大暂停时间不会得到明显改善。

对于对象寿命假说成立的程序而言，因为减少了扫描对象，吞吐量可以得到巨大改善。但是程序的行为，如把对象分成几代，在什么条件下进行重垃圾收集等，会对性能有非常大的影响。

分代垃圾收集利用对象寿命倾向这一性质是一个非常有趣的主意。除引用计数器方式之外，分代方法可以与许多其他垃圾收集结合起来使用，一般与复制方式相结合得比较多。

最近几乎所有的 Java 都实现有分代垃圾收集。另外，函数型语言 OCaml (Objective Caml) 也采用了分代垃圾收集。

14.3.11 保守垃圾收集

像 C 这种本来没有垃圾收集的语言，编译之后没有保留区别整数和指针的信息。因为 CPU 对两者不加区分，所以也就没有这个必要。通常，垃圾收集的实现需要明确区分引用（指针），而 C 和 C++ 没有这样的功能，在这样的环境下实现垃圾收集的技巧之一称为保守垃圾收集 (conservative GC)。

其基本思想就是如果碰巧有整数的值与引用相同的话，该对象就有可能被引用，也就当它是活着的。保守就是倾向于安全的意思。与此相对，能够明确区别引用的环境下的垃圾收集称为精确垃圾收集（exact GC）。

因为倾向于安全一面，保守垃圾收集在 C 或 C++ 这样没有垃圾收集功能的语言中也可以得到实现，这是它的优点，但反过来，本来应该回收的对象却在意料之外保留下来不能回收，这是它的缺点。还有，它不能与复制垃圾收集这样需要移动对象的垃圾收集算法组合使用。

Ruby 采用的是保守垃圾收集。局部变量可以按照通常语言的访问路径来处理，系统堆栈部分是当作指针数组来扫描的。Ruby 大部分是用 C 编写的，因为有了保守垃圾收集，C 库的实现变得非常简单。实际上，Python 和 Perl 因为采用的是引用计数器，C 例程内部的引用数管理非常复杂，偶然忘记增减就会发生内存问题。在这一点上，用 C 编写 Ruby 扩展库的时候，基本上不用操心内存管理，好处十分明显。

Ruby 以外有一个名为 Boehm GC

（http://www.hpl.hp.com/personal/Hans_Boehm/gc/）的库。它为 C 和 C++ 增加了垃圾收集功能。Boehm GC 也同时实现了分代垃圾收集。

Boehm GC 不仅用于 C 和 C++，在 Scheme 处理系统 Gauche

（<http://pratical-scheme.net/gauche/>）等多种语言处理系统中，也作为垃圾收集功能得到广泛的应用。

14.3.12 增量垃圾收集

在实时性要求高的程序中，相对于吞吐量而言，更重视的是最大暂停时间。比如在机器人姿势控制程序中，如果因为垃圾收集而使控制暂停哪怕是 0.1 秒，机器人就会摔倒。或者是在汽车的刹车控制程序中，如果因为垃圾收集而使反应变慢的话，真是不堪设想。

在这类程序中，垃圾收集带来的中断时间必须是可以预测的。比如可能有类似这样的限制条件：即使是最差的情况，垃圾收集也必须在 10 毫秒内完成。

普通的垃圾收集算法都不能保证这一点。因为暂停时间会随着对象数量和状态而改变。为保证实时性，不需要等垃圾收集完全执行结束，

而是要把垃圾收集处理细分成许多细小的片段，每次执行一点，这叫增量垃圾收集。

增量垃圾收集在垃圾收集处理过程中程序也在同时执行，引用有可能会改变。结果是垃圾收集的一致性就不能得到保证。增量垃圾收集为避免这样的问题，采用了与保守垃圾收集相同的写屏蔽技术。

嵌入式处理系统采用的是增量垃圾收集。有名的 Io (<http://iolanguage.com/>) 和 Lua (<http://www.lua.org/>) 都实现了增量垃圾收集。

14.3.13 并行垃圾收集

最新配有多 CPU 核心的多核电脑开始普及起来。不仅是服务器，美国 Intel 公司的 Core 2 Duo 是个人电脑 CPU，使多核电脑变得不再稀奇。

在这样的多核环境下，灵活运用线程可以最大限度地发挥多个 CPU 的能力。并行垃圾收集就是要最大限度地利用多个 CPU 的能力。

并行垃圾收集的基本原理与增量垃圾收集大致是一样的，都是利用写屏蔽来维护当前状态的信息。有的并行垃圾收集的实现生成垃圾收集专用线程，把垃圾收集设计成总是与普通处理并行执行。

美国 Sun 公司的 Hotspot JVM 等实现了并行垃圾收集。

14.3.14 位图标记

以 Linux 为首的 UNIX 系列操作系统在使用 fork 系统调用复制过程的时候，内存空间并不进行复制而是直接共享。

其中任何一个过程要往内存里写数据的时候，操作系统的虚拟内存系统都会捕捉到这一要求，某个页面要写入数据时则复制该页面³。这个写时复制 (copy-on-write) 功能可以减少不必要的页的复制，从而节约内存空间，改善性能。

3 页是虚拟内存管理内存的单位。

不过，垃圾收集与这一为写而复制的功能兼容性不好。给引用对象设置标记的方式会因设置标记而复制包含有该对象的内存页。

复制方式也同样会产生大量写内存的操作。生成子进程处理的程序，会在发生垃圾收集的瞬间引起操作系统复制大量的内存页。利用子进程的程序常常会因为这个内存页复制而引发性能问题。

位图标记是在利用标记的垃圾集中消减操作系统内存页复制的方法。它不在对象里设置被引用的标记，而是利用外部位图区域（管理内存区域）来保存引用标记。

结果，只有标记用的位图部分会发生内存页复制，从而避免了复制没有实际更新的对象所在的内存页。

Ruby 的垃圾收集也有了实现位图标记的补丁⁴。Web 应用程序环境在有的情况下可以改善性能。

4 详细请引用 <http://www.rubyist.net/~matz/20080205.html#p01>。

14.4 用 C 语言来扩展

Ruby 是解释型语言，也就是说，Ruby 程序是由名为 Ruby 解释器的程序来解释执行的。

提起解释器，大家都可能觉得它是逐行读取程序并执行的，实际上 Ruby 解释器是把要执行的程序全部读进来，首先变换成更有效率的内部表现形式。解释器对其内部表现加以分析来执行程序，但从外部看不到 Ruby 程序变换成内部表现的样子。

这是解释型语言的特征，也就是说，程序修改之后，马上就可以照样执行，开发周期非常快。相对于拥有这种性质的解释型语言，C、C++或 Java 这样的编译语言是首先把程序变换成计算机可以直接执行的形式，然后再从头执行。

因为在执行程序的时候，编译型语言不再需要对原来的程序进行解释和变换，大都速度较快。但是，在开发过程中需要有编译这一步骤，而且为了生成高速执行形式，编译器要进行各种各样的处理（称为优

化)，开发周期容易变长。10 多年前，我曾在某公司开发过商业软件，当时曾发生过花了一个晚上还没有编译完的情况。从那以来，计算机性能已经得到巨大改善，我想现在不会再有那样的事情发生了。

14.4.1 开发与执行速度的取舍

像这样执行速度慢但开发周期快的解释型语言，与开发周期容易变慢而执行速度高的编译型语言，常常需要进行取舍。Ruby 的设计方针是开发效率优于执行效率，选择解释型的实现也就是必然的¹。

1 但是，现在有多种 Ruby 执行系统并不都是解释型的。比如 xruby 处理系统是把 Ruby 程序编译成 Java 字节码。

那么，这个 Ruby 解释器（是我开发的，通称为 Matz's Ruby Interpreter，省略为 MRI）又是用什么语言开发的呢？它是用 C 语言开发的²。

2 其他还存在有用 Java 开发的 JRuby，用 C#开发的 IronRuby 以及使用 C++与 Ruby 本身开发的 Rubinius 等 Ruby 处理系统。

采用 C 语言的理由如下：

- 我本来是 C 程序员，C 语言是我最拿手的；
- C 允许系统调用，速度高；
- 用面向对象语言来实现别的面向对象语言的话，容易混淆对象的概念。

实际上，我觉得用 Java、C#或者 C++来开发其他语言处理系统的开发者真是了不起。

Ruby 解释器的构造大致如图 14-24 所示。字句解析器和语法解析器二者结合，构成了把 Ruby 程序转换成内部表现形式的编译器。Ruby 1.8 的内部表现是名为构造树的环状结构。Ruby 1.9 也生成同样的构造树，然后再转换成名为字节码的字节串，这种字节码成为 1.9 版的最终内部表现形式。

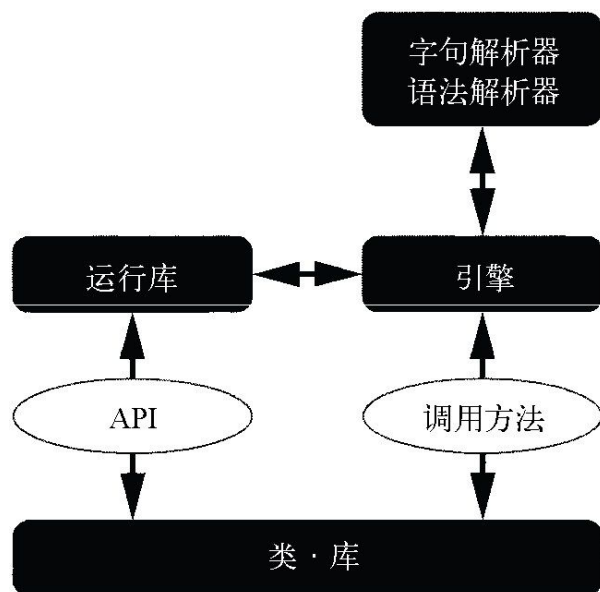


图 14-24 Ruby 解释器的构成

编译器生成的内部表现传递给称为引擎的部分。引擎是解释内部表现，并实际上执行程序的部分。1.9 版的引擎解释的字节码，可以说是近似于虚拟计算机的机器语言，所以 1.9 版的引擎也可以称为 VM，即所谓的虚拟机（Virtual Machine）。

顺便说一下，1.9 版的 VM 的代码是 YARV（读成“雅鲁布”或者“亚布”的人好像挺多的），是 Yet Another Virtual Machine 的缩写。这是因为当初开始开发 YARV 的时候，已经有多个面向 Ruby 的 VM 在开发之中，而 YARV 则作为另一个 VM 而诞生了。结果，当时开发的其他的 VM 都没有能够实现 Ruby 的所有功能，最终正式采用了 YARV。正式采用了之后，就不应该再称为 Yet Another VM，也许应该改称 The VM 了，但因为不忍心扔掉这个长时间叫惯了的具有亲切感的名字，所以也就继续沿用了下来。

闲话休提。引擎在解释内部表现的时候，需要其他组件的帮助。内存和对象的管理，变量访问和方法调用的实现等，都要依靠底层称为运行库的组件来完成。运行库提供底层强有力的支持，是程序执行时不可或缺的部分。

Ruby 利用的各种类是作为类库提供的。Ruby 中的这些部分也是用 C 编写的。像 Ruby 这样的解释型语言，因为是经过 C 编写的引擎来间接执行程序，与一般的编译语言相比，执行速度明显慢得多（100 倍

或者 1000 倍)。但是，实际上程序的执行时间大部分都花在 C 编写的类库方法内部，因此在执行速度上很少会产生那么巨大的差距。

14.4.2 扩展库

扩展库是指利用 Ruby 运行库的 API，用 C 定义类库。Ruby 的 API 使用 C 几乎可以实现 Ruby 所有的功能。使用这些 API 可以很简单地实现如下的功能。

- 定义类。
- 定义方法。
- 访问实例变量。
- 调用方法。
- 调用块。

但是，如果仅仅是要实现与 Ruby 同样功能的话，是没有必要特意用 C 来编写的。Ruby 能做的事情就用 Ruby 来做好了，这样既不需要浪费编译的时间，编写和执行又都简单。

特意花时间用 C 来实现扩展库的理由主要有以下两点。

- 想要比 Ruby 执行速度快。
- 想使用 C 可以利用的库。

前面已经说过，Ruby 是解释型语言，它的执行速度不如像 C 这样的编译型语言。如果是绝对需要改善速度的程序，用 C 扩展库来实现其中成为瓶颈的部分的话，有可能显著改善程序的执行速度。

UNIX 系列操作系统的大多数功能都是通过 C 可以访问的库来提供的。Ruby 要想利用这些功能的话，就需要有一种从 Ruby 调用 C 的 API 的方法，这最简单的方法就是利用扩展库。

14.4.3 看例题学习扩展模块

要学习编程的话，查看实际运行的程序代码是最有效的方法。这次把UNIX的简易数据库 **dbm** 作为例题。Ruby 附带有 **dbm** 接口的标准扩展库，程序共有 829 行，提供有非常丰富和复杂的功能。这次让我们来作一个简单的 **minidb**。因为一点特别的原因，我们不用传统的 **DBM**，而使用 **QDBM** 库。

这次作为例题的 **minidb** 库由 **MiniDB** 类来实现，其中仅仅定义 5 个方法。我认为，就这些已经足够让我们掌握扩展库的基本使用方法了。

MiniDB 类的规格如表 14-1 所示，**MiniDB** 类的用法示例如图 14-25 所示。

表14-1 MiniDB 类的方法

方法名	参数	机能
initialize	DB路径	初始化
[]	键	取得数据
[]=	键, 值	设置数据
close	无	关闭DB
each	无	循环

```
require 'minidb'

db = MiniDB.new("/tmp/foo")
db["abc"] = "123"
p db["abc"]
db["def"] = "456"
p db["def"]

db.each do |k,v|
  p [k, v]
end
db.close
# 输出:
# "123"
# "456"
# ["abc", "123"]
# ["def", "456"]
```

图 14-25 使用 MiniDB 类的例子

让我们以此为基础来编写 **MiniDB** 类初始化部分的程序。扩展库初始化的时候调用如下的函数：

Init_库名

库名一般选择为类名的小写形式，这次就把它命名为 **minidb**。先作成 一个名为 **minidb** 的目录，在这个目录中作一个名为 **minidb.c** 的文件。首先来编写 **minidb.c** 的初始化部分（参见图 14-26）。

```
#include "gdbm.h"
#include <ruby.h>

static VALUE rb_cMiniDB;

Init_minidb()
{
    rb_cMiniDB = rb_define_class("MiniDB", rb_cObject);
    rb_define_alloc_func(rb_cMiniDB, minidb_alloc);
    rb_define_method(rb_cMiniDB, "initialize", minidb_init, 1);
    rb_define_method(rb_cMiniDB, "[]", minidb_get, 1);
    rb_define_method(rb_cMiniDB, "[]=", minidb_set, 2);
    rb_define_method(rb_cMiniDB, "close", minidb_close, 0);
    rb_define_method(rb_cMiniDB, "each", minidb_each, 0);
    rb_include_module(rb_cMiniDB, rb_mEnumerable);
}
```

图 14-26 minidb.c 初始化部分

关于个别的 API 函数这里先不作说明，我们可以从代码中的英文词想象出函数的功能大致应该是这样的：首先定义类，然后定义 5 个方法。**MiniDB** 类有 **each** 方法，顺便把 **Enumerable** 模块包含进来。有了这一行，**MiniDB** 类就可以使用 **Enumerable** 模块提供的大量方法了。

简单说明一下 `Init_minidb` 函数内容，`rb_define_class` 函数是定义类的，参数是类名和父类，它返回新定义类的对象。

`rb_define_alloc_func` 函数指定为对象分配内存的方法。关于这个函数稍后再作说明。

这次虽然没有用到，但顺便说明一下，利用 `rb_define_class_under` 函数可以定义嵌套类。第一个参数指定嵌套类外侧的类或者模块。其余参数与 `rb_define_class` 是一样的。还有，把这些函数名中 `class` 的部分替换成 `module` 的话，就可以定义模块。但是，模块是没有父类的，所以 `rb_define_module` 只有第一个参数。

使用 `rb_define_method` 函数定义方法。第一个参数是拥有该方法的类或者模块对象，第二个参数是方法名，第三个参数是实际上实现该方法的C函数指针，最后第四个参数指定C函数所接受的参数。当第四个参数是正整数的时候，函数接受同样个数的VALUE参数（参见图 14-27a、b）。当参数为-1的时候，函数接受C的数组为参数（参见图 14-27c）。而在参数是-2的时候，函数接受Ruby的数组为参数（参见图 14-27d）。

```
/* a) 第四个参数是0，没有参数 */
VALUE func0(VALUE self) {
    ....
}

/* b) 第四个参数是1，有一个参数 */
VALUE func1(VALUE self, VALUE arg) {
    ....
}

/* c) 第四个参数是-1，接受数组为参数 */
/* 注意参数的顺序是(argc, argv, self) */
VALUE func2(int argc, VALUE *argv, VALUE self) {
    ....
}

/* d) 第四个参数是-2，接受Ruby 数组为参数 */
VALUE func1(VALUE self, VALUE args) {
    ....
}

....
```

```
rb_define_method(Class, "meth0", func0, 0)
rb_define_method(Class, "meth1", func1, 1)
rb_define_method(Class, "meth2", func2, -1)
rb_define_method(Class, "meth3", func3, -2)
```

图 14-27 方法函数取得参数的方法

Ruby 的方法具有可见性，并据此决定什么地方可以调用它。
`rb_define_method` 定义可见性为 **public** 的方法，定义其他可见性（**private** 及 **protected**）方法的时候，分别使用 `rb_define_private_method` 和 `rb_define_protected_method`。这些函数定义的方法仅仅是可见性有所不同，因此参数与 `rb_define_method` 是一样的。

Ruby也可以给个别的对象定义方法（定义特殊方法）。在C的水平上可以使用`rb_define_singleton_method`来定义特殊方法。第一个参数不是类或模块，而是拥有该方法的对象，其他参数与`rb_define_module`都是一样的。

使用 `rb_include_module` 函数在类或模块中包含其他模块。

14.4.4 QDBM函数

这次用到的 QDBM 函数如图 14-28 所示。

```
/* 打开数据库 */
DEPOT *dpopen(const char *name, int omode, int bnum);

/* 从DBM 取得数据 */
/* 一般指定start 为0, max 为-1 */
/* 字符串的长度保存在变量ksiz 里 */
/* 返回的字符串需要free */
char *dpget(DEPOT *depot, const char *kbuf, int ksiz, int start,
int max, int *len);

/* 往DBM 保存数据 */
/* dmode 可以指定下面其中一个 */
/* DP_DOVER - 覆盖 */
/* DP_DKEEP - 不覆盖 */
/* DP_DCAT - 追加 */
int dpput(DEPOT *depot, const char *kbuf, int ksiz, const char
```

```

*vbuf, int vsiz, int dmode);

/* 对数据库中的键开始循环 */
int dpiterinit(DBM *dbm);

/* 返回数据库中下一个键 */
char *dpiternext(DEPOT *dbm, int *len);

/* 关闭数据库 */
datum dpclose(DEPOT *dbm);

```

图 14-28 DBM 函数

把这些函数调用组合起来，就可以实现使用 QDBM 的编程。我们的目的是用 Ruby 来方便地调用这一函数群。

请注意，QDBM 库是以 DEPOT 结构为中心，类似于面向对象的调用。这意味着我们只要把 DEPOT 结构看成是 Ruby 对象就可以了。也就是因为这一点，MiniDB 类的规格才变成前面表中说明的样子。

让我们接着来编写 MiniDB 类的中心内容吧。首先要实现一个封装 DBM 结构的 MiniDB 对象。

像这次用对象来封装 C 结构的情况，需要告诉 Ruby 为对象分配内存的方法。为对象分配内存的函数是 Init_minidb 函数调用的 rb_define_alloc_func()。参数是要指定生成对象的类和生成函数。

对象生成函数 minidb_alloc 的定义如图 14-29 所示。

```

static void
minidb_free(DEPOT *db)
{
    if (db) dpclose(db);
}

static VALUE
minidb_alloc(VALUE klass)
{
    return Data_Wrap_Struct(klass, 0, minidb_free, NULL);
}

```

图 14-29 对象生成函数 `minidb_alloc`

`minidb_alloc` 使用 `Data_Make_Struct` 宏定义来生成封装结构的对象。`Data_Make_Struct` 宏定义需要 4 个参数，它们分别是：

- 对象的类；
- GC 标志函数指针；
- 用来释放结构的函数指针；
- 结构指针。

关于 GC 标记函数，当封装的结构直接或者间接引用 Ruby 对象的时候，会影响到对象管理，GC 标记函数是用来告诉 Ruby 运行库这些参数信息的。这次因为没有这样的引用，所以参数指定为 0。关于标记函数的详细说明，请引用 Ruby 源代码附带的 `README.TXT` 文件，或者《Ruby 编程》（Ohm 出版社）等书籍。

作为用来释放所封装结构的函数，这里指定的是 `minidb_free` 函数。它调用 `dpclose` 函数。但是，这里封装的结构有可能是 `NULL`，所以追加了检查代码。

本来应该把结构指针传递给 `Data_Wrap_Struct` 宏定义，但因为这个时候还没有给要封装的结构（`DEPOT *`）分配内存，所以暂时指定为 `NULL`。

14.4.5 初始化对象

分配了对象之后，就调用它的 `initialize` 方法，这与 Ruby 程序的类是一样的。`initialize` 方法是由 `minidb_init` 函数实现的（参见图 14-30）。

```
static VALUE
minidb_init(VALUE self, VALUE path)
{
    DEPOT *db;
```

```

        SafeStringValue(path);
        db = dpopen(RSTRING_PTR(path),
DP_OREADER|DP_OWRITER|DP_OCREAT, 0);
        if (!db) {
            rb_raise(rb_eRuntimeError, "dpopen - %s",
dperrmsg(dpecode));
        }
        DATA_PTR(self) = db;

        return self;
    )

```

图 14-30 对象初始化函数 `minidb_init`

不管是哪个函数，基本构造都是一样的：

```

(必要的话) 检查参数;
调用处理函数;
出错的时候抛出异常;
返回值。

```

`minidb_init` 函数也不例外。首先，使用 `SafeStringValue` 宏定义来检查参数确实是字符串。类型变换和检查都由这个宏定义来负责完成了。`SafeStringValue` 是用来检查参数是“安全”字符串的宏定义。所谓安全就是，参数不是由外部输入的不能信赖的字符串。如果用不能信赖的字符串作为数据库路径名的话，会带来意想不到的安全性问题，所以这里特意进行了检查。在不需要检查安全性的时候，可以使用 `StringValue` 宏定义。

为了从字符串对象中取出 C 的指针，这里使用的是 `RSTRING_PTR` 宏定义。另外，用 `RSTRING_LEN` 宏定义可以取得字符串的长度。

下一步调用 `dpopen` 函数，打开数据库，得到要封装的结构。

如果在打开数据库的时候发生什么错误的话，就应该抛出异常。要抛出异常可以用 `rb_raise` 函数来实现。第一个参数是异常类，第二个参数以后与 `printf` 是一样的。

最后用 **DATA_PTR** 宏定义来更新一直是 **NULL** 的结构指针，完成初始化处理。一个个看下来，其实也没有什么特别难的东西。

14.4.6 实现方法

接着让我们来看一下其他方法的实现吧（参见图 14-31）。

```
static VALUE
minidb_get(VALUE self, VALUE key)
{
    DEPOT *db;
    char *p;
    int len;
    VALUE str;

    Data_Get_Struct(self, DEPOT, db);
    StringValue(key);
    p = dpget(db, RSTRING_PTR(key), RSTRING_LEN(key), 0, -1,
&len);
    str = rb_tainted_str_new(p, len);
    free(p);

    return str;
}

static VALUE
minidb_set(VALUE self, VALUE key, VALUE val)
{
    DEPOT *db;

    Data_Get_Struct(self, DEPOT, db);
    StringValue(key);
    StringValue(val);

    if (!dpput(db, RSTRING_PTR(key), RSTRING_LEN(key),
                RSTRING_PTR(val), RSTRING_LEN(val), DP_DOVER)) {
        rb_raise(rb_eRuntimeError, "dpput failed");
    }
    return val;
}

static VALUE
minidb_close(VALUE self)
{
    DEPOT *db;

    Data_Get_Struct(self, DEPOT, db);
```



```

    dpclose(db);
    DATA_PTR(self) = 0;
    return Qnil;
}

static VALUE
minidb_each(VALUE self)
{
    DEPOT * db;
    char *p;
    int len;
    VALUE key;

    Data_Get_Struct(self, DEPOT, db);
    dpiterinit(db);
    for(;;) {
        p = dpiternext(db, &len);
        if (!p) break;
        key = rb_tainted_str_new(p, len);
        rb_yield_values(2, key, minidb_get(self, key));
    }
    return self;
}

```

图 14-31 minidb 方法的实现

所有的方法都是首先使用 **Data_Get_Struct** 宏定义，然后取出封装的结构。

接着用 **StringValue** 来进行类型检查，**rb_raise** 来抛出异常，都跟上面讲述的模式是一样的。

需要特别说明的是 **rb_tainted_str_new** 和 **rb_yield_values** 这两个函数。

从数据库中读取出来的字符串要转换成Ruby的字符串。但是，从数据库读取的字符串是从外部输入的，并不一定是可以信赖的。使用 **rb_tainted_str_new** 函数，明确指出它不是安全字符串这一事实。Ruby能够检查出来使用不安全字符串的危险操作，这意味着不容易发生跨站攻击的安全性问题。

rb_yield_values 是Ruby中相当于**yield**的函数。C很难像Ruby那样根据参数的个数而改变动作，所以用**rb_yield_values**来明确

地指明参数个数。只指定一个值的时候也可以用 `rb_yield` 函数。

14.4.7 关于垃圾收集的注意事项

Ruby 有垃圾收集的机制，能够自动回收不再使用的对象。这种机制非常好，但也有需要稍加留意的地方。

Ruby 的垃圾收集属于所谓的保守垃圾收集类，Ruby 的对象如果被 C 的变量访问着的话，就不会被回收，这一点是相当出色的。但是，如图 14-32 所示的情况就会发生问题。

```
static VALUE
foo(VALUE self)
{
    VALUE str = some_func();
    char *p = RSTRING_PTR(str);

    /* 使用 p 的处理 */
    /* p 还在使用之中，str 已不再使用 */
    /* 垃圾收集把str 回收，p 变得无效 */
    /* 严重的程序错误 */
}
```

图 14-32 保守垃圾收集的麻烦问题

这里，引用着 Ruby 对象的变量 `str` 经编译器优化处理之后，已不存在于程序之中，即使 `p` 引用着 `str` 所指向的地址，Ruby 的垃圾收集也不能认识到 `str` 指向的对象还在使用之中这一事实，就会发生这样的问题。

避免这一问题的有效方法是，在成为问题原因的 `VALUE` 变量前加上 `volatile` 修饰。

14.4.8 其他的Ruby API

为了在 Ruby 与 C 之间进行数据交换，Ruby 还提供了其他数量众多的 API，这里就不再逐一详细介绍了。比如，Ruby 提供有如下功能的 API。

- 访问实例变量。
- 定义或引用常数。
- 调用方法。
- 检查和变换各种数据类型。
- 解析方法的参数。

详细请引用 README.TXT 等文件。

14.4.9 扩展库的编译

即使写好了源代码，如果不加编译的话，也是没有意义的。以下是扩展库的编译步骤。

首先按照以上的顺序编写源代码。若 C 程序的文件名后缀一律约定为.c 的话，后续步骤会自动识别出来 C 程序文件。

为生成编译所需要的文件，需要准备必要的 Ruby 文件。这个文件通常命名为 extconf.rb。minidb 的 extconf.rb 的内容如图 14-33 所示。

```
require 'mkmf'

if have_library("qdbm") and
have_header("qdbm/depot.h")
  create_makefile("minidb")
end
```

图 14-33 extconf.rb

extconf.rb 是由以下几个部分构成的：

- 调用 require 'mkmf' ；
- 用 have_library 和 have_header 检查必要的库和头文件是否存在；

- 用 `create_makefile` 来生成必要的 `Makefile.create_makefile` 的参数是库的名字。

照图 14-34 执行 `extconf.rb`，就可以生成 `Makefile`。

```
% ruby extconf.rb
checking for main() in -lqdbm... yes
checking for qdbm/depot.h... yes
creating Makefile
```

图 14-34 生成 `Makefile`

用 `make` 命令来编译，尔后可以用 `make install` 命令来安装编译后的库。

14.4.10 扩展库以外的工具

Ruby 的扩展工具不仅仅是扩展库。这里简单介绍一下其他的 Ruby 扩展工具。

RubyInline

RubyInline 可以在 Ruby 的源代码中直接嵌入 C 的程序。不用每次准备 `extconf.rb` 文件，也不需要明确的编译步骤，非常便利，但它也有不少约束，比如在执行环境中需要有编译器，与外部库的链接也有些麻烦等。

Ricsin

YARV 的开发者笹田先生最近在开发 Ricsin。RubyInline 只是直接利用扩展库的 API，省略了编译的步骤而已，而 Ricsin 把 VM 作为自己的一部分，只有方法处理的一部分采用 C 来实现。这样可以避免方法调用的开销，能够期望提高处理速度。Ricsin 可以从 Ruby 程序库的 `ricsin` 分支得到。

dl

Ruby 程序执行环境，特别是 Windows 环境，并不总是备有编译器。dl 库使得 Ruby 水平的外部库直接调用成为可能。dl 是 Ruby 本身的标准库之一。

有了 dl，仅用 Ruby 也能实现与 minidb 同样动作的库。把图 14-35 的程序保存成 minidb.rb 文件，没有扩展库也可以使用 minidb。dl 的优点是在没有安装编译器和头文件的环境下也可以运行。

```
require "dl/import"
require "dl/struct"

class MiniDB
  module Impl
    extend DL::Importable

    dlload "libqdbm.so"

    extern "DEPOT *dpopen(const char*, int, int)"
    extern "int dpclose(DEPOT*)"
    extern "int dpput(DEPOT*, const char*, int, const char*, int,
int)"
    extern "char *dpget(DEPOT*, const char*, int, int, int, int*)"
    extern "int dpiterinit(DEPOT* )"
    extern "char *dpiternext(DEPOT*, int*)"
  end

  def initialize(path)
    @db = Impl.dpopen(path, 7, 0)
    # 7 = DP_OREADER|DP_OWRITER|DP_OCREAT
  end

  def [](key)
    Impl.dpget(@db, key, key.size, 0, -1, nil)
  end

  def []=(key, val)
    unless Impl.dpput(@db, key, key.size, val, val.size, 0)
      raise RuntimeError, "dpput failed"
    end
  end

  def close
    Impl.dpclose(@db)
    return nil
  end

  def each
    Impl.dpiterinit(@db)
    loop do
      key = Impl.dpiternext(@db, nil)
    end
  end
end
```

```

        break unless key
        yield key, self[key]
      end
    end
  end
end

```

图 14-35 使用 dl 的 minidb

ffi

关于Ruby的扩展，**ffi** 库受到关注。**ffi** 是foreign function interface的缩写，它也提供在Ruby水平上的与外部函数的直接接口，这个目的与**dl** 几乎是一样的。**ffi** 可以从RubyGems得到。

ffi 的特征是使用 **libffi** 更有效率地调用外部函数，Ruby、Rubinius 和 JRuby 都可以使用同样的 API。

使用**ffi** 库实现**minidb** 的程序如图 14-36 所示。

```

require 'ffi'

class MiniDB
  module Impl
    extend FFI::Library
    ffi_lib("qdbm")
    attach_function "dpopen", [:pointer, :int, :int], :pointer
    attach_function "dpget", [:pointer, :pointer, :int, :int,
    :int, :pointer], :pointer
    attach_function "dpput", [:pointer, :pointer, :int, :pointer,
    :int, :int], :int
    attach_function "dpclose", [:pointer], :int
    attach_function "dpiterinit", [:pointer], :int
    attach_function "dpiternext", [:pointer, :pointer], :pointer
  end

  def initialize(path)
    @db = Impl.dpopen(path, 7, 0)
  end

  def [](key)
    len = MemoryPointer.new(:int)
    d = Impl.dpget(@db, key, key.size, 0, -1, len)
    s = d.read_string(len.read_int)
    d.free
    s
  end
end

```

```

end
def [] = (key, val)
  unless Impl.dpput(@db, key, key.size, val, val.size, 0)
    raise RuntimeError, "dpput failed"
  end
end

def close
  Impl.dpclose(@db)
  return nil
end

def each
  Impl.dpiterinit(@db)
  loop do
    len = MemoryPointer.new(:int)
    p = Impl.dpiternext(@db, len)
    break if p.null?
    key = p.read_string(len.read_int)
    yield key, self[key]
  end
end
end

```

图 14-36 ffi 实现的 minidb

* * *

本节学习了 Ruby 的扩展功能。使用 Ruby 的扩展 API 可以比较简单地实现 C 的高速化库或外部库的接口。另外，使用 `dl` 或 `ffi` 这样的技术，不用 C 就可以实现对外部库的访问。

14.5 为什么要开源

我与开源/自由软件也打了很长时间的交道了。最初接触到自由软件还是大学时代 1989 年的事了。编辑器 Emacs 和编译器 GCC 是我最初接触到的自由软件。

我自己最早发布的自由软件是 *Emacs Lisp* 一书中介绍的邮件阅读器 `cmail`，已经记得不是很清楚了，我想那大致是 1990 年前后的事情。在那之后，又发布了 Ruby（1995 年），跳槽到现在的公司后，专门从事自由软件的开发（1997 年），在我的生活中，开源所占的比例越来越大。

沉浸在“开源生活”中的我也有困惑的时候。对于 IT 行业以外的人士，以及即使是 IT 行业内但对开源保持一定距离的人士，我都一直很难让他们明白“开源是什么”这个问题。因为我也有自己的立场，不愿意使用简单而又容易产生误解的“免费软件”这种说法，即使费劲说了一大堆，结果听的人还是一付不知所云的面孔。

这种“难以理解度”好像与对“自由软件”这个单词本身的误解，以及它在不同侧面所拥有的多种概念不无关系。

14.5.1 自由软件的思想

首先，让我们从开源与自由软件的区别开始吧。按照出现顺序，首先是 1980 年前后出现了自由软件一词，后来（1998 年）才诞生了开源软件一词。

自由软件意味着用户可以自由处理的软件，它的背景是软件是自由的思想。

自由软件这一思想起源于自由软件基金会（Free Software Foundation, FSF），关于软件的自由，它是这样考虑的：

- 执行的自由
- 阅读源代码的自由
- 改变源代码的自由
- 再发布源代码的自由

为保证执行的自由，必须能够将软件免费拿到手。出于学习和研究的目的，也必须能够自由地得到源代码。还有，为了修改程序错误，或者为适合自己的目的而进行改造，那就不单单是阅读源代码，还要允许改变源代码。把自己认为好的软件推荐给别人，或者把自己进行的修改或改善与他人共享的话，就需要有再发布的自由。

这些自由并不是自然发生的，而是需要我们大家来保护和培育，这就是 FSF 的主张。这也有其历史根源。

14.5.2 自由软件的历史

曾几何时，在计算机的黎明期，软件完全是硬件的附属物。买了计算机，附带操作系统等源代码，根本就不是什么稀罕的事。那时，或者在更早一些的时候，甚至有这样的情况，从生产厂家买来的计算机连操作系统也没有，需要用户自己来开发各种软件。买了同样计算机的用户互相交换自己开发的软件，互助合作。虽然不够精致，但与如今开源软件的情况很有些相似之处。

但是，随着时代的发展，软件成了商品，源代码也摇身一变，成为不能简单对外开放的企业机密。对于就这么一路走过来的人而言，就相当于逐渐剥夺了他们的自由。1970 年以后，软件的源代码就不再是理所当然可以拿到手的东西了。

这一背景与当时美国微软公司的总裁比尔·盖茨先生写的《致业余爱好者的一封信》有关，这一事实并不广为人知。在发表于 1976 年 2 月的这封信中，盖茨先生提出，只有业余爱好者能开发出高质量的软件吗？专业人员得免费工作吗？他要求把软件作为商品来处理。

即使这样，大学里的人们还在继续以一种田园牧歌的方式进行软件开发，但这一自由逐渐被商业软件所侵蚀。

14.5.3 Emacs事件的发生

随之发生了具有象征意义的事件。成为舞台的就是现在还在广为使用的高性能编辑器 Emacs，主要登场人物是 FSF 创始人、麻省理工学院的天才程序员理查德·斯托曼（Richard Stallman）先生，和后来因设计 Java 而成名的詹姆士·戈斯林（James Gosling）先生。

最初的 Emacs 是斯托曼用宏编辑器 Teco 编写的。作为易于使用的编辑器而受到好评的 Emacs 有众多的派生版本。1981 年，后来成为 Java 设计者的戈斯林开发了 UNIX 版的 Emacs。

戈斯林开发的 Emacs（通称 GoslingEmacs 或 Gosmacs）拥有使用名为 MockLisp 的类 Lisp 语言开发的扩展功能。斯托曼也想要 UNIX 版的 Emacs，希望能在戈斯林版的 Emacs 基础上进行开发。他想要的

Emacs 不是 MockLisp 这种类 Lisp 语言，而是融合了真正 Lisp 的 Emacs。

但是，戈斯林把 Gosmacs 的专利卖给了 Unipress 公司，斯托曼就不能够在 Gosmacs 的基础上开发新的 Emacs 了。结果，斯托曼从零开始开发了自己的 UNIX 版的 Emacs。

也就是在这个时候，他开始意识到，软件的自由并不是哪儿都有，而必须自己来保护。这时可以说，意识到“只是公开源代码并不够”这一事实的斯托曼比大众领先了 15 年。

在那之后，斯托曼成立了保护软件自由的团体 FSF，定义了保证软件自由的许可证 GPL（GNU General Public License）。另外，他还劝说其他人也采用 GPL 软件许可证，参加保护软件自由运动。这些是 1983 年的事。

从那时以来，斯托曼与 FSF，在 GPL 下公开了大量的软件。优秀的编译器 GCC，以及涵盖了 UNIX 大多数命令的 GNU 工具等，FSF 的工具已成为开源中不可或缺的存在了。他们的最终目标是，创造一个从上到下完全自由的操作系统环境 GNU（GNU's Not Unix），而 GNU/Linux¹ 的出现几乎达成了这一目标。

1 按照斯托曼的主张，Linux 是内核的名称，各种 GNU 工具结合而成的操作系统应该称为 GNU/Linux。

但是，自由软件并没有得到企业的理解。一种说法是自由这个词让人联想到免费，还有的说斯托曼仇恨商业软件的态度也让经营商业软件的 IT 企业对他敬而远之，搞不清楚到底是怎么回事。

14.5.4 开源的诞生

在这样的环境中，身负把自由软件推广到商业领域重任而诞生的单词是开源。

1998 年，在浏览器战争中败给微软公司的美国网景公司决定把自己开发的 Web 浏览器的源代码作为自由软件公开。其背景是埃里克·雷蒙德（Eric Raymond）先生发表的论文《大教堂与集市》。

这篇论文发表于 1997 年的 Perl Conference。这篇论文以 Linux 为题材，考察了通过公开源代码，采用不同于像建筑大教堂那样的基于精密设计和计划而执行的传统软件开发方式，而是像集市一样宽松由志愿者自发合作的方式，从而促进优秀软件的诞生。

网景公司受到这篇论文的强烈影响。为了打开局面，他们作出了一个划时代的经营策略，决定公开自己公司的主要产品 Netscape Communicator 浏览器源代码。之后，雷蒙德等自由软件界的主要人物作为顾问，与网景公司的管理层举行了战略性会谈，当场决定开创公开源代码和商业联姻的新模式，为了适当地表达这一概念，“开源”一词诞生了。

当时开源意味着公开源代码，只是现在开源的各种特征中极为有限的一小部分。因为开源这个词中不包含自由这个最为重要的概念，受到斯托曼等自由软件界人士的反对。开源差点就因为他们的反对而流产了。

但是，采用开源这一新词有如下的好处：

- 新词带来新气象；
- 不再过分强调免费这一不符合商业要求的侧面；
- 给人一种好像新商业趋势的印象。

开源一词因此得到商业人士的支持。针对自由软件派的“轻视自由”的反对意见，现实派并不怎么重视这一点，认为开源的定义（OSD）保证了自由。

在开源一词诞生之后，以雷蒙德为中心的组织 Open Source Initiative（OSI）马上给开源下了一个明确的定义。根据这一定义，开源软件就是用满足开源定义的许可证所发布的软件。开源定义的概要如图 14-37 所示。

请注意，开源定义既非作者的意向，也不是软件本身的性质，而是用许可证来定义的。也就是说，开源并不关心软件本身，而只是关心如何使用和发布软件。

传统软件的许可证总是在限制用户的权利，与此相反，FSF 定义的 GPL 则保证自由软件的软件自由，保护用户的权利，在这个意义上，它是划时代的。开源的定义比使用许可证来判断一个软件是否是自由软件更进了一步。开源的定义是以 Debian GNU/Linux 发布中为判定什么样的软件是自由软件而制订的 DFSG（Debian Free Software Guideline）为基础的。

1. 可以自由再发布
2. 可以得到源代码
3. 可以存在衍生作品，衍生作品可以使用同样的许可证
4. 在允许发布补丁文件的时候，可以要求保持完整性
5. 不歧视个人和团体
6. 不歧视应用领域
7. 再发布时没有必要追加许可证
8. 不依赖于特定的产品
9. 不影响同一媒介发布的其他软件
10. 技术上保持中立

图 14-37 开源的定义

DFSG 是为了把大量的软件整理成一个发布时，如何判定自由软件而制订的。如果是满足 DFSG 许可证的软件，那么在发布和改变之前就不再需要每次征求开发者的承诺，可以安心地制作发布版本。

可以说继承DFSG而诞生的开源也是一样的。满足开源软件，也即 OSD 许可证而发布的软件，在开发方面基本上没有什么限制，可以安心地开发和发布。看了 OSD，也有不少人觉得条件多限制多，既然是开源，仅仅公开源代码还不够吗？但在发布软件集合时，能做的事情是对个别软件所能做事情的最大公约数，如果不加一定的限制，能做的事情就会越来越少。作为开源特征的集市开发也是一样，为保证开发工作的顺利进行，有必要使用适当的许可证来让自己安心。

14.5.5 OSS许可证

我们已经明白，开源定义的实质在于开源许可证的条件，对于自由软件/开源软件来说，许可证是至关重要的。许可证明确规定了用户可以怎样使用和发布该软件，对于自由软件来说，它是思想的表现，或者是保护自由的武器。

满足开源定义的许可证有很多。在 OSI 的网页上 (<http://www.opensource.jp>)，经 OSI 认可的许可证就多达 72 种。还有没经 OSI 认可，但也满足开源定义的许可证，所以其总数还会更多。

Ruby 的许可证也没有经 OSI 认可。如今进军开源领域的企业逐渐增加，拥有法律部的大企业，好像大都盲从于律师而制定自己独特的许可证。实际上，OSI 认可的许可证中，相当大一部分都冠有美国苹果公司、美国 IBM 公司等一些公司的名字。

OSS 许可证中最有影响的有以下几种：

GPL

GPL (GNU General Public License) 应该可以说是自由软件许可证的鼻祖，它具有如下特征：

- 没有保证；
- 表示版权；
- 保持同样的许可证。

GPL 特征中最重要的是保持同样的许可证，也就是说，在对 GPL 许可证的软件进行修改或复制的情况下，必须保持其原来的 GPL 许可证。

这意味着再利用 GPL 软件的源代码而开发的软件的许可证也必须是 GPL。批判 GPL 的人把这称为感染性。赞同 GPL 的人们称之为版权保留，因为这一性质意味着版权 (copyright) 的保留。

GPL 的最新版本是 3.0。这一版本发表于 2007 年，其中国际法及有关专利的条款比以前的版本更为明确。GPL 软件的 GPL 3.0 移植好像也在顺利进行之中。也有像 Linux 这样的软件，明确表示将继续使用原来的 GPL 2.0。

LGPL

LGPL (GNU Lesser General Public License) 也是与 GPL 拥有同样性质的许可证，但对于适用 LGPL 许可证的软件库，如果只是链接在一起

的话，就不能做到版权保留，所以 **LGPL** 把保持同样许可证的适用范围只限定为对该软件的修改。这是因为如果对链接库也全部适用 **GPL** 许可证的话，就有可能限制库的使用范围。

LGPL 本来只是针对库而制定的，原名为 **GNU Library General Public License**，**FSF** 为了表明这仅仅是例外情况，不应该被盲目使用，因而从保证自由的观点考虑，把它改名为 **Lesser**（次）。

BSD许可证

加利福尼亚大学伯克利分校在公开 **BSD UNIX** 时，制定的许可证是 **BSD** 许可证。当初 **BSD** 许可证由以下三项组成：

- 没有保证；
- 保持版权表示；
- 衍生产品的广告中介绍原作者（宣传条款）。

但是 **BSD** 有以下缺点，由于最后称为宣传条款的项目，在包含有大量软件的软件包时，有可能导致介绍原作者的部分会比广告内容还要长的情况，它比 **GPL** 的限制还要严格，**GPL** 许可证的软件与 **BSD** 许可证的软件的源代码不能混合使用（**GPL** 非互换性）。

在那之后，经过加利福尼亚大学的同意，删除了宣传条款，现在只有前两项的修正 **BSD** 许可证得到广泛使用。

MIT 在公开 **X Window System** 时采用的许可证通称为 **MIT** 许可证（也称为 **X11** 许可证），其内容与修正 **BSD** 许可证几乎是一样的。

BSD 许可证与 **MIT** 许可证没有感染性，在希望自己的软件得到更广泛使用的层次中很受欢迎。另一方面，与 **GPL** 相比，**BSD** 因保护（强制）自由的力量较弱而受到批判。

APL和CPL

其他许可证中比较有名的有 **Apache** 基金会采用的 **Apache** 许可证（**APL**），**IBM** 提倡的 **CPL**（**Common Public License**）等。前者不是版权保留，后者是版权保留。

这些许可证的特征是考虑到了专利和商业利用等情况。

14.5.6 开源的背景

知道了开源的定义和历史，还不能说就理解开源了。最重要的问题是为什么要开源，或者为什么开源能行得通，这些问题也还没有得到充分说明。

与 1976 年盖茨提出的“专业人员得免费工作吗”的疑问相反，有很多人积极参加开源软件的开发工作。其中既有把自己的时间自发地无偿贡献出来的人，也有把开发开源软件作为职业的人。这种与直觉相反的情况是如何发生的呢？

在科学领域，共享知识和信息本来是非常普通的事情。即使不用看站在巨人的肩膀上的牛顿的例子，把知识作为论文（免费）公开，利用前人的成绩进行新的研究是再理所当然不过的了。

软件，特别是商业软件，一旦作为商品来经营的话，就容易忘记这样一点，与论文的内容一样，软件也不过是信息的一种，也应该适用同样的原则。科学家致力于研究，大学或研究机构对科学家给予支持，这种体制对软件也有可能成立。实际上，大学或研究机构支援软件开发的例子相当多。

但是，谈到近年来开源的普及和发展，计算机的普及和因特网的发展则功不可没。过去如果要想开发高质量软件的话，需要购买价格高昂的计算机，召集大量的技术人员。现在一般家庭里的计算机都完全能够开发软件，通过网络进行合作开发的事情也屡见不鲜。个人出于兴趣而编程也已经能够达到相当高的水平了。

关于自由软件世界，斯托曼在他写的《GNU 宣言》中，对未来做出了如下的预言。

“从长远的观点看来，程序员成为自由程序员，是走向完美世界的第一步，在那样一个世界里，谁都不用再为维持生计而像奴隶一样劳动。人们每星期只要工作 10 个小时，比如在完成制定法律、家族交流、机器人修理或小行星探测等必要的工作之后，就可以自由地专注于编程这一充满乐趣的活动。再也不用为维持生计而编程了。”

人类还没有达到不用为维持生计而编程的程度，但比 100 年前还是富裕得多。好像有很多人在业余时间里沉浸于编程的乐趣之中。因此，可以理解的是，开源激发了优秀程序员们的创造力，刺激了他们对知识的好奇心，使他们从中得到至高无上的快乐。另外，可以感觉到关心软件自由的人也在增加。

想知道软件是如何执行的，就去阅读源代码。软件有错误的时候，就自己来修改。如果觉得自己行动受到限制的话，就阅读源代码，利用编程技术来排除这一限制。这些都是程序员的本能。

软件的复杂化和商品化也是开源的背景之一，不容忽视。软件所覆盖的领域越来越广，软件也越来越复杂。过去 1 万行左右的程序就实现了的操作系统，如今变成了超过 600 万行的巨大软件。

14.5.7 企业关注开源的理由

不过，人们钱包里的钱并没有发生什么明显的变化，即使是每个软件的规模变得越来越大，软件整体的投资并没有增加。结果，软件开发的预算急剧减少，除非是特别大的软件公司，一些小公司已经无法仅靠自己来开发和提供人们所需要的软件。

这就是企业关注开源的理由。从 1998 年以来，开始出现了盈利企业为自己的利益而开发开源软件的事例。一旦认识到无法自己来开发所有软件之后，企业只自己开发最具竞争力的小部分核心软件，而让大家共同来开发其他软件，结果对大家都有好处，企业关注开源正是这种冷静的分析考量的结果。

另外，对于采用这一战略的公司而言，积极参与开源软件开发，拥有开源开发人员，可以证明自己公司的技术实力，有助于确立品牌。我工作的网络通讯研究所即是这样一家企业，此外，还有不少通过这一战略而取得成功的企业。

参与开源的人们也各有各的想法。有因经营考虑而参加的企业，有为软件自由而参加的程序员，有因上级命令而参加的公司职员，各种各样，千差万别。但是，开源有可能为个人和全人类带来幸福，如果能继续下去形成良性循环的话，那真是再好不过了。

14.5.8 Ruby与开源

Ruby 从一开始就是开源软件。Ruby 在 1995 年初次公开的时候，开源这个单词还没有诞生，也许应该称为自由软件。

在这继续开发 Ruby 的 16 年中，我经常会被问到“为什么把 Ruby 开源了呢？”。好像是在说，既然 Ruby 被评为优秀软件，如果商品化的话，我即使不会变成盖茨那样，也会变得相当富裕的。

非常抱歉，有违大家的期望，我除了把 Ruby 作为自由许可证软件公开以外，没有考虑过其他的选择。其理由之一就是，我从大学以来所处的教育环境，几乎全是由以 GNU 软件为主的各种自由软件构成的。

我所使用的编程工具，过去是，现在也是，几乎都是以 Emacs 为主的各种自由软件。还有，我的编程知识也大都是通过阅读自由软件源代码而学到的。

考虑到我的出身，除了服从业务命令而进行的开发之外，把自己开发的软件作为自由软件公开，当然是再自然不过的了。

进一步而言，现在 Ruby 被评为优秀语言，也是因为它公开为自由软件，得到了很多人的帮助。16 年间，很多人为了改进 Ruby 付出了巨大的努力。虽然 Ruby 是我开发的语言，但如果没有大家的力量，Ruby 是不可能今天的。

如果我想“Ruby 是优秀的语言，应该用它来赚钱”，自己创业开公司的话，恐怕现在 Ruby 已经消失得无影无踪了。实际上，过去也有很多企业曾挑战过“语言生意”，但几乎没有成功的，编程语言的生意就是这么难做。

另一方面，因为我没有直接选择商业之路，而是把 Ruby 作为自由软件公开，现在自己的时间几乎 100% 都可以用在 Ruby 上，而且所得到的收入也足以养家。即使不会像盖茨那样富有，开发开源软件也足以维持生计。

14.5.9 选择许可证的方法

如果从此开始开源项目开发的话，那么应该如何为软件选择许可证呢？

许可证与技术无关，而是有关法律和合同的工作，这不是技术人员的工作，而是属于律师的工作范围。对于软件开发人员来说，这决不是一件令人感兴趣的工作，有人甚至会想：“哪用得着这些东西呢？”

那么为什么要特意准备许可证呢？其主要理由是要表明自己的主张。合适的许可证可以明确地表明开发人员希望如何使用和发布该软件，这样用户可以安心地使用软件。

如果这一点不明了的话，用户为了能够真正安心地使用软件，就会直接询问开发人员。对开发人员来讲，如果从世界各地传来大量“这样的情况也可以使用吗”的疑问，这决不是什么令人高兴的事。还有，人们也希望通过许可证来避免诉讼等法律问题。因此，许可证虽是件琐事，却能够让开发人员专注于软件开发。因为这一点，最初的选择也就变得非常重要。

我能给出的首要建议就是，绝对不要自己定义新的许可证。16年前，在开始开发 **Ruby** 的时候，如果能明白这一点的话，我的人生就会快乐许多。当时，我想提供一个明确允许再利用源代码的许可证。就这样，我以 **Perl** 许可证为基础，明确定义了一个允许源代码的再利用，不对输入输出数据（**Ruby** 环境下的 **Ruby** 程序）作限制的许可证。

定义许可证跟编程序也有类似之处。在考虑允许什么禁止什么的时候，跟考虑算法是一样的。现在回想起来，当时还是蛮高兴的。

但是，要消除这个“代码”中的错误，那比程序可要难得多。它不像程序那样可以简单地“执行”，不能马上发现问题，即使发现了问题，修改起来也绝非易事。

例如，假设自己定义的许可证所包含的条款中有点什么问题。马上可以想到的一点就是，因为与 **GPL** 没有互换性，所以不能与 **GPL** 软件链接在一起。基于类似的思想而定义的许可证，却没有互换性，这是一件多么令人伤心的事情啊，但许可证就是这样一个现实。在这种情况下，要么按照字面的意思放弃与 **GPL** 软件的链接（就会有用户感到不方便），要么就来修改许可证。

如果使用你的许可证的软件的所有代码都完全是由你自己写出来的话，这几乎不成为什么问题。只要发布新的许可证版本，问题就解决了。

但是，如果其中包含了其他人写的补丁的话，软件就不再是你一个人的了，在法律上写补丁的人是共同拥有版权的。

严格说来，当然不能随便侵害他人的版权，所以从理论上讲，许可证的修改需要得到所有共同作者同意。像 **Ruby** 这样经过多年开发的软件，已经无法确认到底有多少人拥有相关版权了。

像 **FSF** 软件那样，不管是谁，只要从他那里接受（**10** 行以上的）代码贡献，就必须签订书面的版权转让合同的话，就不太容易发生版权问题。但是，每次收到补丁的时候，都要进行这样的手续是件非常麻烦的事。大多情况下，我只好放弃许可证的修改，或者经过巨大努力之后也无法取得全员的同意就修改了许可证，然后从心底里祈祷没有人会对此抱怨。

自己定义许可证的话，就意味着有好几年的时间，你都要亲自承担这些麻烦事。我从心底里建议你不要定义自己的许可证。

世界上存在有数量众多的开源许可证，从中选择一个就足够了。而且建议选择那些著名的许可证。

发生需要修改许可证才能解决问题的，大都是选择了不怎么出名的许可证的项目，这是众所周知的事实。著名的许可证因为得到广泛的使用，基本上没有遗留什么大问题，即使有什么问题，朋友们大多也安心。不出名的许可证大都没有能够充分考虑到与其他许可证组合的情况，让人不安。

在选择软件许可证的时候，首先要考虑是否希望版权保留。版权保留，简单说就是，不允许那些不拥护自由的人在你写的自由软件上“免费搭车”。

FSF 作为软件自由的拥护者，强烈推行版权保留。另一方面，不怎么在乎版权保留的开发人员好像更多一些。如果你希望版权保留的话，几乎就只能选择 **GPL**。

如果你的软件是作为库来使用的话，那么 **LGPL** 则是个不错的选择。如果对库适用 **GPL** 许可证的话，那事实上就只能为 **GPL** 软件所用，采用限制较为宽松的 **LGPL** 的话，有可能使你的软件得到更为广泛的使用。但是，**LGPL** 有不方便、难于理解以及没有得到充分考察等缺点，在不太强烈希望版权保留的时候，最好不要选择它。

对于不太在乎版权保留的人来说，可以选择 **GPL** 以外的许可证。这里的要点是，该软件是否需要与其他软件进行链接。如果预想到将来可能以某种方式与 **GPL** 许可证的软件链接在一起的话，那么与 **GPL** 的互换性就变得很重要。拥有插件功能的软件或库特别要注意这一点。作为与 **GPL** 不相矛盾的许可证，有修正 **BSD** 许可证和 **MIT** 许可证等。

另一个应该考虑的要点是，与相关软件和许可证是否一致。比如 **Eclipse** 插件就应该选择 **Eclipse** 许可证（即使不与 **GPL** 互换）。另外，**PHP** 关联软件选择与 **PHP** 一致的许可证也是安全的。用 **Ruby** 编写的软件与 **Ruby** 本身的许可证本来是相互独立的，但好像也大都选择 **Ruby** 许可证。

* * *

开源是为了软件自由，在自由软件运动中诞生的。开源这个词是进军商业领域的市场营销用语。公开源代码好像会破坏商业软件，实际上有许多好的运用方法。为了运用得好，许可证的选择就非常重要。

范型的变化

从诞生以来有超过 15 年了，**Ruby** 一直在不断地茁壮成长。速度更快，功能更多，使用更方便，代码更简洁，不知道 **Ruby** 会进化到什么程度。回头看看这几年 **Ruby** 的变化，引人注目的是，它越来越朝支持函数式编程风格的方向发展。

以 **Haskell** 为首的函数式编程语言变得出名，人们能实际感觉到它的便利之处，这是这种变化的原因之一，但其更重要的原因是我个人对函数式编程语言所持的矛盾印象。

像本章已经说明的那样，函数式编程具有非常多的优点。静态声明式的编程代码容易理解，容易检查出错误，类型检查功能也有效。

最重要的是，没有副作用的函数式编程与将来会越来越重要的并行编程的配合简直是天衣无缝。仅就这一点来说，我觉得将来编程的主流很有可能是函数式的。但是，用函数式编程开发一定规模以上的软件时，会突然变得特别难，这也是事实。这也可能是因为程序员还没有习惯函数式风格吧。与结构化编程和从中发展而来的面向对象编程相比，我怀疑从上到下用函数式风格开发一定规模的软件，可能不符合人类的思考风格。

这一怀疑是否正确我们还不知道，就表面看来，亲和性好、效率又高的 **Ruby** 语言借用了函数式语言的思想，如果能把新的函数式编程的优点和友好的面向对象编程的优点结合在一起，那就再好不过了。

也许编程风格的急剧转变马上就近在眼前了。